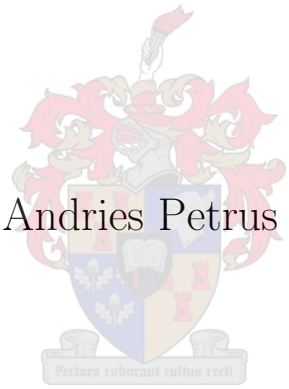
# Merging Deep Neural Networks and Probabilistic Models using Sum Product Networks




Andries Petrus Smit

Thesis presented in partial fulfilment of the requirements for the degree of Master of Engineering (Research) in the Department of Electrical and Electronic Engineering at Stellenbosch University

STUDY LEADER: Prof. J.A. du Preez

DATE: March 2020

# Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Date: March 2020

# Acknowledgements

I would like to acknowledge my study leader, Prof. J.A. du Preez, and my family for their contribution and support throughout this project. I would also like to thank Praelexis© for their generous sponsorship of this research. I also gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used in this research.

# Abstract

In recent years there has been renewed interest in machine learning algorithms that can explicitly model uncertainty. Machine learning has great potential to revolutionise almost every sector of our world. To apply these algorithms in areas such as healthcare, insurance, and other high-risk sectors, it is necessary to know both when they are uncertain and, at least partially, be able to explain their predictions. A doctor, for example, can only accept or reject a potential treatment if they can understand why the machine learning system has made the recommendation. Probabilistic models have attractive properties in this regard, as they provide a wide range of probabilistic queries, which help to better understand the model's predictions. However, these probabilistic models are normally either limited in their predictive accuracies or have slow inference times. Sum Product Networks (SPNs) have been proposed as a promising type of deep probabilistic network, as they enable probabilistic queries to be answered in tractable time while also being expressive with high modelling accuracies. In this work, we investigate how SPNs can help bridge the gap between black-box deep learning models and interpretable but limited probabilistic graphical models. We also investigate learning algorithms for SPNs, and derive a new structure learning algorithm for constructing a complete SPN directly from data in both the generative and discriminative settings.

# Uittreksel

In die afgelope paar jaar is daar 'n hernieude belangstelling in masjienleer-algoritmes wat uitdruklik onsekerheid in hul voorspellings kan modelleer. Masjienleer het groot potensiaal om byna elke sektor van ons wêreld te verbeter. Om hierdie algoritmes in gebiede soos gesondheidsorg, versekering en ander hoë-risiko sektore toe te pas, moet hulle kan weet wanneer hulle onseker is, sowel as ten minste gedeeltelik hul voorspellings kan verduidelik. 'n Dokter kan byvoorbeeld slegs 'n moontlike behandeling aanvaar of verwerp as hy/sy kan verstaan waarom die masjien-leer stelsel hierdie aanbevelings maak. Probabilistiese modelle het aantreklike eienskappe in hierdie opsig, aangesien hulle 'n wye reeks probabilistiese vrae kan antwoord, wat help om die model se voorspellings beter te verstaan. Hierdie probabilistiese modelle is egter normaalweg óf beperk in hul voorspellings se akkuraatheid, óf hulle het baie stadige inferensietye. Die Som Produk Netwerk (SPN) is onlangs as 'n belowende soort diep probabilistiese model voorgestel, waar probabilistiese vrae in 'n redelike tyd beantwoord kan word, terwyl dit ook ekspressief is met 'n hoë modellering akkuraatheid. In hierdie werk ondersoek ons hoe 'n SPN gebruik kan word om te help om die gaping tussen die swartkassie-diepleermodelle en verduidelikbare, maar beperkte, probabilistiese grafiese modelle te oorbrug. Ons ondersoek ook leeralgoritmes vir 'n SPN, en verkry 'n nuwe struktuur-leeralgoritme vir die konstruksie van 'n volledige SPN direk uit data in die generatiewe asook diskriminerende mode.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Nomenclature

### Abbreviations

| | |
|---|---|
| AC | Arithmetic Circuit |
| CNN | Convolutional Neural Network |
| Compress-SPN | Compress - Sum Product Network |
| ConvSPN | Convolutional Sum Product Network |
| DNN | Deep Neural Network |
| DP | Dynamic Programming |
| DSPN-SVD | Discriminative Sum Product Network - Singular Value Decomposition |
| EBW | Extended Baum-Welch |
| EM | Expectation Maximisation |
| Explain-SPN | Explain - Sum Product Network |
| Fashion-MNIST dataset | Fashion - Modified National Institute of Standards and Technology dataset |
| GDA | Gaussian Discriminant Analysis |
| GMM | Gaussian Mixture Model |
| HMM | Hidden Markov Model |

| | |
|---|---|
| LearnSPN | Learn Sum Product Network |
| LSE | Log Sum Exponential |
| MNIST dataset | Modified National Institute of Standards and Technology dataset |
| MPE | Most Probable Explanation |
| PGM | Probabilistic Graphical Model |
| RAT-SPN | Random Tensorized - Sum Product Network |
| SET-SPN | Search, Expand and Tune - Sum Product Network |
| SPN | Sum Product Network |
| SPN-SVD | Sum Product Network - Singular Value Decomposition |
| SVD | Singular Value Decomposition |
| tSPN | Tensor Sum Product Network |

**Greek symbols**

| | |
|---|---|
| $\alpha$ | Learning rate used by the gradient descent optimiser in each weight update. |
| $\hat{\boldsymbol{\theta}}_m$ | Best parameters for the given model $m$ that best describe the data. |
| $\boldsymbol{\theta}_m$ | Parameters of a model. |
| $\boldsymbol{\theta}^*_m$ | Updated parameters of an SPN for the given model $m$. |
| $\lambda$ | Constant that indicates how much the optimiser should focus on regulating the network. |
| $\mu$ or $\boldsymbol{\mu}$ | Mean value or vector of a Gaussian distribution. |

| | |
|---|---|
| $\sigma$ or $\Sigma$ | Standard deviation or covariance matrix of a Gaussian distribution. |

### Other symbols

| | |
|---|---|
| $\mathbb{1}_k(z)$ | Indicator function that outputs 1 if discrete random variable $z$ is at state $k$, otherwise outputs 0. |

### Roman symbols

| | |
|---|---|
| $\mathbf{c}(i+1, j_p)$ | Vector that represents all the $j$ indices of children of node $(i+1)j_p$. |
| $c_{kn}$ | Addition constant value of node $k$ for data point $n$. |
| $c_p(i+1, j_p, j_c)$ | Function that returns the index value in vector $\mathbf{c}(i+1, j_p)$ where the entry value $j_c$ is located. |
| $D$, $S$ and $R$ | Hyperparameters used to generate a RAT-SPN. |
| $g_{kn}$ | Gradient value of node $k$ for data point $n$. |
| $g_k(\sim z_i)$ | Value indicating how much the network output will change for changing the output of $\mathbb{1}_k(z_i)$. |
| $\mathbf{h}$ | Set of hidden random variables in a network. |
| $J$ | Loss of a network for a given dataset. |
| $K$ | Number of probability terms that are being summed or multiplied with each other. |
| $k_y$ | Shape parameter of the Weibull distribution for class $y$. |
| $L$ | Number of leaf distributions in the network. |
| ln | The natural logarithmic function with base $e$. |
| $M$ | Maximum ln probability input to the log sum exponential function. |

| | |
|---|---|
| $m$ | The specific model structure. |
| $\hat{m}$ | The model that best describe the data, where each model has an assigned set of parameters $\boldsymbol{\theta}_m$. |
| $N$ | Number of data points. |
| $n$ | The data point index. |
| $N_e$ | Number of node expansion tests conducted. |
| $N_I$ | Number of layers in an SPN. |
| $N_n$ | Number of nodes in the network. |
| $N_s$ | Number of sum nodes in the network. |
| $N_z$ | Number of random variables being modelled. |
| $p_g$ | Value in the range $[0, 1]$ that indicates how generatively the network should be trained. |
| $p_{kn}$ | Probability output of node $k$ for data point $n$. |
| $p_k(\mathbf{x}, \mathbf{y})$ | Probability output of node $k$ defined over $\mathbf{x}$ and $\mathbf{y}$. |
| $p_l$ | Probability output of a leaf node at index $l$. |
| $p_m$ | Probability distribution over model $m$. |
| $p_r$ | Probability output of the root node of an SPN. |
| $p(z_1)$ | Probability output of random variables $z_1$ given that all the other random variables have been marginalised out. |
| $p(z_1, z_2)$ | Probability output of an example network modelled over random variables $z_1$ and $z_2$. |
| $r_{ln}$ | Value indicating how responsible the leaf node at index $l$ is for representing data point $n$. |

| | |
|---|---|
| $s_y$ | Scale parameter of the Weibull distribution for class $y$. |
| $t_y$ | Offset parameter of the Weibull distribution for class $y$. |
| $v_l$ | Leaf distribution of an SPN at index $l$. |
| $v_{ln}$ | Leaf distribution probability value at index $l$ for data point $n$. |
| $w_{ij}$ | Weight value of sum node $i$ connected to its child node $j$. |
| $w_k$ | Weight value at index $k$. |
| $w^*_{ijk}$ | Updated weight value of an SPN at indices $ijk$. |
| $\mathbf{x}$ | Set of input random variables in an SPN. |
| $\mathbf{x}_{1:N}$ | All the data point values for the random variables in set $\mathbf{x}$. |
| $\mathbf{x}_n$ | Data point $n$'s state values for the $\mathbf{x}$ random variables. |
| $\mathbf{y}$ | Set of output random variables for which we want to infer probabilities. |
| $\mathbf{y}_{1:N}$ | All the data point values for the random variable set $\mathbf{y}$. |
| $\mathbf{y}_n$ | Data point $n$'s state values for the random variable set $\mathbf{y}$. |
| $\mathbf{z}$ | Set of random variables. |
| $z_k$ | Random variable at index $k$. |
| $Z(p_o, g)$ | Normalising function that determines if the model is trained discriminatingly or generatively. |

# Chapter 1

# Introduction

## 1.1 Motivation

Most modern deep learning algorithms generally focus on black-box function approximation. These deep learning methods usually have good classification accuracy, but struggle with providing explanations for their predictions. This explainability problem has hindered the integration of machine learning into many high-risk environments such as healthcare. Probabilistic models, another class of machine learning algorithms, provide more of an explanation for their predictions but are usually not as accurate as neural networks. Sum Product Networks (SPNs) have recently been proposed to help alleviate this problem due to their dual status as a special kind of neural network as well as a probabilistic network. SPNs have fast inference times while still delivering relatively high classification accuracies on large datasets. As SPNs are probabilistic, they can provide more flexible query capabilities and better explanations for predictions than other deep neural networks. It is therefore of interest to investigate the training and leveraging of SPNs' attractive properties to potentially integrate more machine learning into real-world environments.

## 1.2 Background

### 1.2.1 Deep neural networks

The field of deep learning has in recent years achieved state-of-the-art results in an increasingly large number of machine learning challenges. Modern deep learning can arguably be said to have started with the publishing of a famous paper on deep-belief networks (Hinton *et al.*, 2006). However, the main breakthrough that led to its widespread adoption was not until 2012, where a convolutional neural network (CNN) architecture named AlexNet (Krizhevsky *et al.*, 2012) won the ImageNet challenge. This was the first time a neural network won the Imagenet challenge and in doing so, achieved an error rate of 15.3 %, more than 10% better than the second-placed algorithm.

These networks work by performing operations on layers of nodes called neurons. The first layer receives input and performs a non-linear operation on that particular input. The second layer then performs a non-linear operation on the first layer's output. This process is repeated for every layer using the previous layer as input until all the layers have been calculated. The output of the network is then presented at the final layer's output. The operations that are performed are dependent on the weights of the network. By tuning these network weights through a process called backpropagation, further discussed in Section 5.2, the network can be trained to model a specific function, e.g. producing labels for images in Imagenet. Deep Neural Networks (DNNs) are usually considered 'black-box' models as it can be difficult to decipher why these algorithms make the predictions they do, despite research having been conducted in this area (Montavon *et al.*, 2018). Neural networks usually have high classification accuracy, but only provide one probabilistic query - for example, predicting the output label probability for a given input in the case of image classification. If one is only interested in classification accuracy, neural networks are usually a good fit for a given problem. In some high-risk areas, such as healthcare, this limited query capability has, however, hampered their adoption, as a wrong prediction can be life-threatening. Neural networks form part of a broad class of machine learning algorithms, called function approximators. Probabilistic models, another set of machine learning algorithms, try to model a probability distribution over a set of random variables, which usually allows for more flexible inference capabilities.

### 1.2.2 Probabilistic models and inference

The goal of a probabilistic model is to model a joint distribution over a set of random variables. Contrary to function approximators such as neural networks, probabilistic models inherently model uncertainty. After a probabilistic model is constructed, the model can answer a range of different probabilistic queries. These models can also be trained from data, where the model attempts to represent data in a compact form by exploiting independence in the data. Probabilistic models also provide uncertainties over their estimates, which helps one to understand the confidence a model has in its predictions. By performing different probabilistic queries, one can start to understand why a model is making a certain prediction. One method of doing this is by marginalising (removing) input features. There are, however, drawbacks with probabilistic models having all these additional inference capabilities. These models are usually not as accurate on large real-world datasets as for example neural networks are. Alternatively, they can be made to be more accurate, but their inference times (time to answer probabilistic queries) usually increase substantially. The question then is if we can strive for the accuracy and fast inference of a neural network architecture while still having the flexibility of a probabilistic model. This leads us to investigate tractable probabilistic models, discussed in the next section, and later SPNs as a possible solution.

### 1.2.3 Tractable probabilistic models

A probabilistic model is only useful if the model can answer a probabilistic query accurately and in a reasonable time-frame. It is thus tempting to work with probabilistic models that we know have tractable inference once the network structure is constructed. With tractable inference, we mean that exact inference can be done in time that increases linearly as the size of the network increases. In this work, we refer to time that increases linearly as the size of the network increases as time linear to the size of the network. Some guaranteed tractable probabilistic models include graphical models with low treewidth and mixtures of tractable models, like Gaussian Mixture Models (GMMs). A problem with some of these tractable models is that they usually do not yield high accuracy on large real-world datasets (Levray & Belle, 2019). A new addition to these tractable models is SPNs. SPNs can achieve relatively high accuracies thanks to their

ability to recursively build from smaller tractable distributions. They can be seen as a deep-mixture model over tractable leaf distributions.

## 1.3 Literature synopsis

### 1.3.1 SPNs as a new tractable model

SPNs have emerged as a promising type of probabilistic model due to the fact that they are a tractable, specific type of deep neural network that still retains probabilistic querying capabilities. As with deep neural networks, SPNs consist of different layers of nodes that perform operations on the previous layer's outputs. An example of an SPN can be seen in Figure 1.1.

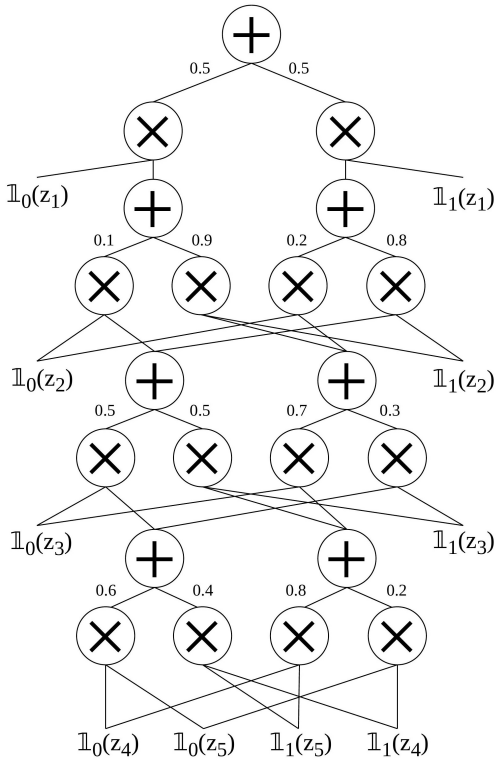



*Figure 1.1: An example of a valid SPN configuration. Note that sum nodes have weights associated with them whereas product nodes do not. Here the $z$ symbols represent discrete random variables. Note that $\mathbb{1}_k(z_j)$ is an indicator function, which outputs 1 if $z_j = k$ ($z_j$ at state $k$) and 0 if not.*

By performing a forward pass through the network, a probability value is calculated and presented at the root (top) node. The sum and product nodes in each layer represent the computations the network has to perform in that layer. The sum and product nodes in the network can have any arbitrary number of children, as long as the network remains a valid SPN. This validity of an SPN is further discussed in Section 3.4.2. Sum nodes have weights associated with them that are first multiplied with that particular node's child probability values before being summed together. Each sum node's weights sum to one and represent a mixture model, with the corresponding weight for every child signifying the importance of that child to the node's output. Product nodes can be thought of as representing local independence assumptions in the network as they multiply together distributions of a disjoint set of random variables. Input random variables can be set to specific values corresponding to the probabilistic query one wants to compute. In Section 3.3 we will discuss the mathematical description of the indicator function, $\mathbb{1}_k(z)$, in more detail. The rationale behind this structure of an SPN is also provided in Section 3.3.

Once an SPN is defined, any joint or marginal probabilistic query over the random variables modelled by the SPN can be answered in time linear to the size of the network. This is a very rare property for probabilistic models, especially if the model has relatively high modelling accuracy, e.g. the testing accuracy of the model. SPNs are in many cases not far behind pure neural networks in classification accuracy (Van de Wolfshaar & Pronobis, 2019), while also retaining probabilistic querying capabilities. One of the reasons SPNs can achieve high model accuracies is due to their ability to model local conditional independence extremely well. This means that SPNs can find independence between random variables even though the random variables are not completely independent. Random variables must usually be approximately independent from one another over all the data in order for probabilistic models like Probabilistic Graphical Models (PGMs) to utilise independence. SPNs can effectively find independence between random variables on a subset of entries in a dataset, which allows for more compact models to be created from noisy real-world data. An example of this local independence is given in Section 3.3.

A important focus of SPN research at the moment is on increasing the training accuracy of these models, by designing better training algorithms, as seen in Peharz *et al.* (2018), Jaini *et al.* (2018) and Van de Wolfshaar & Pronobis (2019). Benchmarks that are normally used for SPNs are those that are commonly used to test other neural network

architectures. Training these SPNs is an active area of research with an increasing number of proposed methods and not one agreed-upon standard yet. Many of these methods are also directly being transferred from deep neural network literature.

### 1.3.2 Training SPNs from data

#### 1.3.2.1 Parameter learning

Parameter learning is a method of training SPNs where the structure of these networks is already defined. The parameters or weights of the sum nodes in the network are then iteratively updated using a parameter learning algorithm. There has been extensive work in the SPN literature on parameter learning in the generative and discriminative settings. These methods rely on first defining a dense random SPN, and then iteratively tuning the parameters in the network to attempt to increase the likelihood of the data given the network. Gradient descent methods were proposed (Gens & Domingos, 2012) and successfully used to train large models by Peharz *et al.* (2018). Expectation Maximisation algorithms have also been proposed (Desana & Schnörr, 2016) for the fast training of SPNs in the generative setting. A parameter learning algorithm (Rashwan *et al.*, 2018), which borrows techniques from the Extended Baum-Welch (EBW) algorithm used in Hidden Markov Models (HMMs), has also been used to train networks in the discriminative setting.

#### 1.3.2.2 Structure learning

Structure learning in SPNs involves learning both the structure and the parameters of the network directly from data. This is generally posed as a harder problem than just performing parameter learning. An upside to structure learning methods is that they do not require random initial networks with possibly bad assumptions on which nodes are connected. There have been a few proposed generative structure learning methods, with recent papers including Hsu *et al.* (2017), Dennis & Ventura (2017b) and Kalra *et al.* (2018). A generative learning algorithm tries to learn a model that represents the entire joint probability of the data over all the random variables provided. In the discriminative training setting, the network is always given some of the random variables but does not need to explicitly model them, and needs only to do inference on the other set of random variables. Image classification, where the model predicts the probability

of each label given the input, is an example of discriminative learning. (Relatively few published papers concentrate on the discriminative structure learning setting.) One of these algorithms uses Singular Value Decomposition (SVD) as presented in Adel *et al.* (2015). This SVD method tries to approximate discriminative structure learning by finding a subset of the input variables that has the strongest correlation with the output label. Generative structure learning is then performed on this subset of the input variables with the expectation that the discriminative capabilities of the network will be higher than performing generative training on the full input variable set. In this work, we also develop a more general algorithm that can directly optimise the network in either a discriminative or generative setting.

## 1.4 Project objectives

The goals of this research project are listed below.

- Determine what capabilities SPNs have as a deep probabilistic network. The study should also include capabilities SPNs have that other deep neural networks and probabilistic models individually do not have.
- Study methods to make SPNs more interpretable (or explainable) so that these models can be better integrated into real-world environments.
- The main challenge facing SPNs is determining an efficient way of generating a valid SPN from data. SPNs have specific structural requirements, which complicate learning. Part of this work investigates and attempts to implement a new general learning algorithm for generating an SPN directly from raw training data. The subgoals are:
  - Investigate the advantages and limitations of current learning algorithms used to generate an SPN completely from data.
  - Attempt to design a new learning algorithm by combining different aspects of current learning algorithms. This learning algorithm should be simple to use, and work in, a wide range of training settings. This algorithm should also be able to work in the generative and discriminative training settings, and be able to work with continuous, discrete and hybrid datasets.

  - Measure how well the different SPN learning algorithms (including the newly developed algorithm) perform on a benchmark dataset.
  - Generate a comparative study of the results obtained in the previous objective.
  - Compare SPN accuracies on benchmark datasets against other machine learning algorithms. These machine learning algorithms should include function approximators such as neural networks and other probabilistic models.

## 1.5 Contributions

The contributions made in this thesis are listed below.

- An SPN library was created from scratch in the Python programming language. Additional code was written for the learning algorithms and tested using the SPN library.
- A study was conducted on the capabilities of SPNs and their unique advantages compared to other probabilistic models and neural networks. This study found that SPNs have guaranteed inference times linear to the size of the network while still being able to yield high modelling accuracies on datasets. It is rare for a probabilistic network to possess both these properties. SPNs can easily work with missing inputs and they have flexible query capabilities thanks to their probabilistic nature, which is usually not possible with other types of neural networks.
- A new learning algorithm called SET-SPN (Search, Expand and Tune - SPN) was created. SET-SPN combines aspects from different SPN learning algorithms. A new method of conducting fast structural search is also introduced in this work and forms part of the core of the SET-SPN algorithm. This algorithm works on most types of data and removes the need to generate a random starting network structure.
- A new fast compression algorithm called Compress-SPN was created in this study. Compress-SPN reduces the size of a network by over 50% on average, without loss in the network's accuracy on the dataset.

- A new multi-configuration inference algorithm was also derived for SPNs, which allows for multiple probability values to be inferred from the network with the same computational cost as calculating one probability value. This algorithm forms the basis of the Explain-SPN algorithm, as is discussed next.
- A new explainability algorithm called Explain-SPN was also created. It can generate simple human-interpretable explanations of the predictions made by the network. It was found that, using the Explain-SPN algorithm, SPNs can indeed generate rudimentary explanations. This is the first known algorithm designed for SPNs to generate explanations for its predictions.
- A study on learning algorithms used in SPNs was conducted and we investigated which learning algorithms achieve the best results. In this study it was found that the SET-SPN algorithm achieved the best discriminative accuracy on the MNIST dataset. Whereas better results have been published for the RAT-SPN and ConvSPN algorithms, one advantage that SET-SPN offers over the other learners is that it creates compact models with fast inference times. We thus propose using the SET-SPN algorithm on a given problem first and then tuning the weights of the generated network using the Adam optimiser, which is used in the RAT-SPN algorithm.
- We also generated and compared training results of SPNs with other machine learning algorithms on the Fashion MNIST and MNIST datasets. It was found that SPNs performed better than the other probabilistic models tested, but were less accurate than neural networks (as expected).
- The ability of SPNs to work with missing data was evaluated and their results were compared to neural networks. Here we investigated SPNs' classification accuracies on the MNIST dataset when only some of the inputs were given. It was found that SPNs outperformed a standard neural network in this training setting.

## 1.6 Overview

In this work, we are interested in evaluating SPNs and their attractive properties for real-world applications. Neural networks are known for their fast and high predictive

accuracies, but usually can only answer queries with fixed inputs and outputs. In contrast, probabilistic models are extremely flexible but either struggle with exact inference or cannot model data to a high level of accuracy. SPNs can overcome some of these hurdles as they are probabilistic models with exact inference while still having good modelling accuracy. They achieve this by approximating a distribution during training and not during inference, as some other probabilistic models do. In this work, we investigate learning algorithms and the capability of SPNs in order to determine how well they can be integrated into real-world environments.

### 1.6.1 Training of SPNs

An open challenge that remains in SPN research is to prove that they can achieve relatively high accuracies, compared to the state-of-the-art results usually achieved by other neural network architectures, although recent results are getting close (Van de Wolfshaar & Pronobis, 2019). Created in 2011, the SPN is a relatively new type of model and the training of these networks is still an active area of research. The number of articles on training SPNs has grown appreciably in recent years. In Chapter 2 we discuss a number of popular, recently-proposed learning algorithms. We also subsequently design a new learning algorithm, in Chapter 6, called SET-SPN that can easily be adapted for a wide range of training settings. In Section 7.2.3 we evaluate these different learning algorithms used for SPNs. We evaluate the accuracies of the different learning algorithms using the MNIST dataset. The MNIST dataset is a classification dataset, where one is provided with an input image and must predict the digit represented in the image. The MNIST dataset was chosen as the evaluation dataset as almost all the SPN learning algorithms have published MNIST results. It is thus easier to compare our results to those obtained by the original authors of each publication on a learning algorithm. This dataset and another more challenging dataset, called Fashion MNIST dataset, are further discussed in Section 7.1. The Fashion MNIST dataset is also a classification dataset, but instead of classifying digits, as in the MNIST dataset, one has to classify articles of clothing. This classification is posed as a harder problem to solve than the vanilla MNIST dataset. In our implementations of some of the SPN training algorithms, the SET-SPN achieved the highest discriminative accuracy on the MNIST dataset, while also creating a compact network. However, some MNIST results for SPN learning algorithms from other studies

have achieved higher scores than the results reported in this study. We propose using the SET-SPN algorithm in cases where one desires a model to be compact while achieving relatively high accuracy. The SET-SPN algorithm is also extremely flexible and can thus be used out of the box on most datasets.

In Section 6.2, we also developed a fast compression algorithm that we call Compress-SPN. This algorithm is designed to reduce the size of the network without decreasing the network's modelling accuracy. We find that Compress-SPN on average reduces the size of SPNs upwards of 50% with negligible loss to accuracy. The only compression algorithm we know of that compresses a network better is the tSPN algorithm. However, this compression algorithm creates a tensor version of the SPN, which makes it impractical to update the network further. It is recommended to first use the Compress-SPN algorithm while the network is still training, as is done in SET-SPN. After the network is trained and no changes need to be made to the network again, the tSPN algorithm can be used for final compression.

Lastly, we evaluate the classification accuracies of SPNs compared to other types of models, described in Section 7.2.4, on both the MNIST and Fashion MNIST datasets. Here we compare SPNs to a deep neural network as well as two types of probabilistic models. It was found that deep neural networks achieve the highest raw classification accuracies, with SPNs in second place, above the other types of probabilistic models. As we will see in the next section, however, when not all the input random variables are known, SPNs achieve competitive results compared even to deep neural networks.

### 1.6.2 Capabilities of SPNs

In Chapter 4, we investigate different inference capabilities of SPNs. Along with exact inference, SPNs are also probabilistic models. These probabilistic properties enable SPNs to be able to work with partially observed inputs. Before using the network to compute a probability value, one first sets the random variables to their specific state values. Each random variable that is set to a specific state is said to be observed. If the state of a random variable is not known, it is said to be unobserved. By simply setting all the indicator nodes of the unobserved random variables to 1, as described in Section 4.4, the network effectively marginalises out these random variables. Thus any marginalised probability value can also be calculated in time linear to the size of the network. In

Section 7.2.5, we evaluate the predictive accuracies of SPNs when not all the inputs are given and find that they outperform a standard neural network. This is true even though the neural network has a higher fully input-observed classification accuracy than the SPN. This result shows that SPNs can indeed work well in environments where predictions need to be made with only a limited subset of inputs observed.

A final important property that these SPN models need to have is to be able to explain why they made a certain prediction. This is needed to help experts in a field build trust in these systems, and to help integrate these machine learning models into society. In Section 4.7.2 we investigate how these SPNs can be used to generate explanations for their predictions by using a multi-configuration inference technique. This multi-configuration inference technique, described in Section 4.6, allows for more that one selective probability value to be inferred in time linear to the size of the SPN. To our knowledge this is the first time this inference algorithm is adapted to work in SPNs. In Section 4.7.2, we subsequently develop an algorithm, called Explain-SPN, that can generate explanations that a human can interpret. This Explain-SPN algorithm is evaluated in Section 4.7.2.2 by tasking it with indicating which pixels in an image are most important to its claims. The network is tasked with predicting which digit (0-9) is present in an image and also indicating which pixels best predict that digit. By evaluating the images generated by the algorithm, one can conclude that it is indeed focussing on the correct pixels. The system can also be made to generate explanations for predictions it does not consider likely, which can be useful in cases when the system's best prediction might be wrong. We also see in Section 4.7.1 that SPNs can also generate new data samples. By looking at these data samples a person can better start to understand what the network is focussing on in making predictions, especially in the discriminative setting.

In the next chapter, we investigate what research has already been conducted on learning algorithms and inference capabilities in SPNs.

# Chapter 2

# Literature study

## 2.1 Probabilistic models

### 2.1.1 Introduction

A probabilistic model works according to underlying rules described in probability theory, which is a branch of mathematics that specialises in modelling random phenomena. These models can be used to predict unknown and future events. If one can represent and perform inference on a probability distribution, a wide range of probabilistic queries can be answered about the variables being represented. This is in contrast to function approximators, which can usually only answer one type of query. Probabilistic models are used to represent probability distributions for two main reasons, namely tractable inference and generalisation. Compact forms of probability distributions are needed to make inference on these distributions tractable. Generalisation means that a model should not try to fit its training dataset as well as possible, but should instead try to create a model that will perform well on the unseen data on which it will be used. The network should thus only model the underlying pattern in the provided dataset and not model the irrelevant noise of that data, which does not help with predictions on new unseen data. Probabilistic Graphical Models (PGMs) are one type of probabilistic model. In PGMs a compact representation of a probability table is used to make representing and calculating values from this joint probability table tractable (Koller & Friedman, 2011). These compact forms can be achieved due to conditional independence in the random variables being modelled. There has also been work on training PGMs directly

from noisy real-world data (Zhou, 2011). If one works from noisy data, the compression capability of a model allows for a better representation of the underlying patterns in the data. Thus the model generalises better. To achieve these compression capabilities on noisy data, these models first need to be trained from that data, as discussed next.

### 2.1.2 Training probabilistic models

A probabilistic model can be trained in a generative or discriminative setting. In the generative setting, the model attempts to represent the entire joint probability distribution of the data over all the random variables. These models are usually used to generate new data points such as, for example, new images of similar objects to those it was trained on. They are, however, less accurate at representing conditional distributions, than models specifically trained to do so, using discriminative learning. In the discriminative setting, the model attempts to maximise its ability to predict some output label given a set of input random variables. An example of a challenge in this training regime is in classifying the object present in an image. In the discriminative setting, the model still represents a joint probability distribution, but it represents one that better describes the discriminative features in the data. The model, therefore, does not try to represent the whole original joint probability distribution, as there is additional information that the network does not need for classification. This also means that the network cannot necessarily accurately generate new data. In this work, we investigate a class of tractable probabilistic models called Sum Product Networks (SPNs). SPNs are preferable to some other probabilistic models in that they are fully probabilistic but still have inference times linear to the size of the network. We next discuss some previous work on SPNs.

## 2.2 Overview of sum product networks

Probabilistic models have a reputation for being extremely expressive and useful for relatively small problems but impractical to implement for environments with a large amount of data. Here we refer to the expressiveness of a model as the range of distributions a model can represent, e.g. the different types of data that can be accurately represented by a model. Probabilistic models are usually either expressive in the number of distributions they can represent with no guarantee in terms of their inference speeds, or they have fast

inference but are not very expressive. In Poon & Domingos (2011), the authors try to address this problem by creating a new type of probabilistic network. They investigate under what general conditions a probabilistic network can be created that can represent a wide range of distributions while also having fast and exact inference. This led them to discover SPNs.

As the name suggests, SPNs are networks that consist of sum and product nodes. These sum and product nodes represent the computation that is performed in the network. Computations are performed from the bottom of the network upwards until the root (top) node is reached. This root node represents the probability output of the query made on the network. SPNs can be seen as an exponentially deep mixture model, where sum nodes represent adding different weighted mixture components together. The weights of each sum node add to one, and represent the importance of each mixture component to the sum node's probability output. A product node represents local independence between random variables.

As SPNs are a probabilistic model, they represent a joint probability distribution. The difference between SPNs and most other probabilistic networks is that probabilistic queries performed on any SPN have inference time linear to the size of the network, while still being expressive. One therefore does not have to consider inference time when designing a network so long as the network is not extremely large. The challenge that is now presented is learning an accurate model from the data. It is thus of interest to investigate different algorithms already proposed to train SPNs from data, as discussed next.

## 2.3 Learning algorithms for SPNs

As previously described, a probabilistic network can be trained to represent data in two ways, namely generative and discriminative training. A mathematical description of generative and discriminative learning is provided in Section 3.2. There are also two random variable types that SPNs work with, namely continuous and discrete random variables.

The goal of a learning algorithm for SPNs is to maximise the likelihood of some data for a given network. In other words, we want to find a network that is most likely to produce the data provided. Usually, a prior probability distribution over possible

networks is also included to make sure the network is likely to generalise and not overfit on training data.

In addition to being able to train an SPN in the generative and discriminative settings, there are two methods of actually updating the network. The first method is called parameter learning. Parameter learning usually begins with a random starting network. The learning algorithm then attempts to learn the weights for this network that best fit the data. The network's weights correspond to the weights of each sum node. Therefore, the structure of the SPN remains unchanged in parameter learning. The second and more general learning regime in SPNs is called structure learning. Structure learning algorithms, which are an important part of this work, learn the whole SPN from data. We also focus on parameter learning, as it is used in many structure learning algorithms to fine-tune the network's weights. In structure learning, the structure and weights of the network are completely learned from data. Due to this added difficulty of also learning the structure, structure learning is considered a harder problem than parameter learning. Work that has been conducted on both these learning regimes is discussed next.

### 2.3.1 Parameter learning

As mentioned at the start of this section, parameter learning is only concerned with learning the weights of the sum nodes in the network. Thus a randomised dense structure is usually generated beforehand for the parameter learning algorithm to work from.

The first two parameter learning algorithms were proposed in Poon & Domingos (2011). These two methods are used to do generative parameter learning in SPNs and are called gradient descent and Expectation Maximisation (EM).

#### 2.3.1.1 Gradient descent

Gradient descent is used extensively in training deep neural networks. The gradient descent algorithm attempts to directly move the weights in the direction that minimises the loss function of the network. The loss function, usually defined in terms of the negative likelihood of a network, is a function that if minimised is predicted to present the best network to use for future predictions. We would like to change each weight slightly in the direction that reduces the loss of the network. If we can achieve this slight reduction in loss, this process can be repeated to keep reducing the loss of the network.

To apply gradient descent, we first calculate the first-order gradient of the loss function with respect to each parameter in the network. The real loss function is a complex, high-dimensional function. Therefore, there is no known exact method for finding the network parameters that minimise this loss function, and an iterative method is needed. The first-order gradients of the loss with respect to a parameter indicate in which direction a parameter should be changed to reduce the loss of the network. However, these gradient values are usually only an accurate indicator of the direction and magnitude to update the network's parameters for small changes in those parameters. In other words, for small updates to each parameter in a network, the gradients are reliable enough to predict how the loss of the network will change. The value predicted to decrease the loss, which is used to update each parameter, is proportional to the negative of the gradient. The parameters of the network can now be updated by first calculating the gradients of the loss function with respect to each parameter and then updating each parameter. By repeating this process, the network's loss will start to decrease as the network starts to better fit the data. This method was successfully used to train SPNs on image completion (Poon & Domingos, 2011) using generative gradient descent. In Gens & Domingos (2012), the gradient descent algorithm was also adapted to do discriminative parameter learning. The gradient descent algorithm can also easily be used to train networks with continuous and discrete random variables.

#### 2.3.1.2 RAT-SPN

In Peharz *et al.* (2018), the authors used standard deep neural network methods to train a randomly generated dense SPN. This SPN is called a Random Tensorized - SPN (RAT-SPN). They used the PyTorch library with a more advanced gradient descent optimiser called the Adam optimiser. Furthermore, they also implemented regularisation, which helped the learning algorithm to generalise better. They achieved an MNIST testing accuracy of 98.19%, which was significantly higher than previous parameter learning results and generally high for a network architecture that is not specifically shaped to work with image data.

#### 2.3.1.3 Conv-SPN

An expressive convolutional SPN architecture, or ConvSPN, was published in Van de Wolfshaar & Pronobis (2019) and Cory J. Butz & Teixeira (2019). For this work, the authors tried to transfer techniques for Convolutional Neural Networks (CNNs) to SPNs. Directly transferring techniques from CNNs to SPNs is generally difficult. This is due to SPNs having strict structural requirements for them to remain probabilistic. Van de Wolfshaar & Pronobis (2019) achieved state-of-the-art results for SPNs on MNIST image classification, with an accuracy of 99.19% at the time of publication.

#### 2.3.1.4 Expectation maximisation

The second parameter learning method proposed in Poon & Domingos (2011) is called Expectation Maximisation (EM). Expectation Maximisation is a standard algorithm used in probabilistic models to find the parameters of a network. EM is used for training SPNs in the generative setting. The EM algorithm consists of two parts, namely the expectation step and the maximisation step. The algorithm works iteratively, starting from an initial random network. In the expectation step, all the nodes in the network are assigned responsibilities for representing different parts of different data points. The algorithm estimates what probability distribution each node should represent given the current network state. In the maximisation step, the weights of the network are updated so that each sum node best represents the data distribution according to the responsibilities it was assigned. The network should now be slightly better fitted to the data, which increases the likelihood of the network. This process can now be repeated until the likelihood of the data given the network is high enough or a maximum training time has been reached.

### 2.3.2 Structure learning

In contrast to parameter learning, structure learning learns the complete structure of the network directly from data. The reason why complete structure learning is considered at all is due to the importance of the structure of an SPN to the final likelihood of the data given the network. In neural networks, every subsequent layer can be fully connected to the previous layers. This allows the neural network to be able to compactly represent a

wide range of functions by only changing the weights of the network. SPNs cannot be fully connected to each subsequent layer. The initial structural setup of an SPN therefore has a greater effect on the distributions the SPN can model. Unfortunately, it is harder to transfer techniques for neural networks on how to do structure learning due to the specific structure of SPNs. Most of the structure learning methods used for SPNs are therefore newly developed, as we will discuss next.

#### 2.3.2.1 LearnSPN algorithm

The first top-down structure learning algorithm was proposed in Gens & Domingos (2013). This algorithm, called LearnSPN, is used to construct a complete SPN from data in the generative setting. The LearnSPN algorithm is recursive. The algorithm takes as input a data matrix and then attempts to find independence between the random variables using this data. If it does find independence, it creates a product node to represent that independence in the network. If no independence is found, the data is clustered and a sum node is created instead. The amount of data assigned to each child of a sum node determines the weighting of the particular child node. Every child node of the new sum or product node then again applies the LearnSPN algorithm on the subset of the data assigned to the particular child. A network resulting from a dataset is illustrated in Figure 2.1. In Figure 2.1 the same colour for a given random variable means the same state value for that random variable. Colours are used to illustrate that there can be entries that are similar to one another in a larger dataset.

In Figure 2.1 the data is first clustered into 3 clusters, as there is no initial independence present. The yellow and blue cluster then gets split into two independent distributions, signified by the product node. LearnSPN uses a G-test of pairwise independence to test if random variables are independent in the data provided. The G-test is a mathematical function that can score two discrete random variables based on how similar they are to each other. Two random variables are considered independent if their G-test score is below a certain threshold value. This structure learning algorithm only works in the generative setting and has a fixed execution time for a fixed dataset.

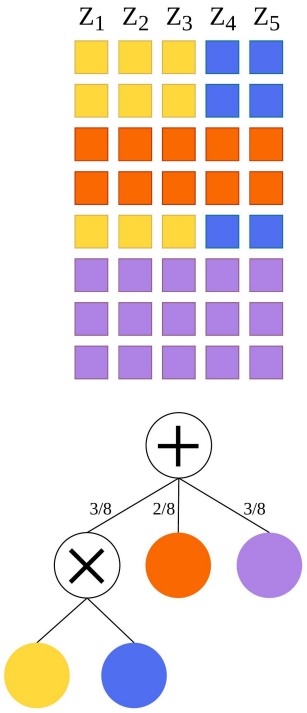



*Figure 2.1: Illustration of how an SPN would look for a given data example using LearnSPN. Each colour signifies a specific state values for the random variable. Note that no independence is initially found and the data first needs to be clustered.*

If one has an environment where new data is being added as time progresses, this algorithm would not be suitable to use.

#### 2.3.2.2 SPN-SVD algorithm

Adel *et al.* (2015) proposed an improvement to the LearnSPN algorithm by using Singular Value Decomposition (SVD) instead of clustering and independence testing. This algorithm, called SPN-SVD, recursively extracts rank-one sub-matrices from the data provided to it. This method relies on SVDs to compute the singular values needed in the update rule of the SPN-SVD algorithm. The algorithm tries to find sub-matrices that are large, but still approximately rank-one. A rank-one sub-matrix can be represented by a small number of nodes, otherwise called leaf distributions. These rank-one sub-matrices are seen as the basic building blocks, otherwise called the atoms, from which

the algorithm builds a network. What this means is that the algorithm finds, in the provided data matrix, the largest sub-matrix that can be described by a small number of leaf nodes. The algorithm therefore attempts to greedily represent the most data it can with the smallest number of nodes it can. The algorithm splits the data matrix into three smaller matrices, namely the approximately rank one sub-matrix, the data subset matrix 1 and the data subset matrix 2. This process is illustrated in Figure 2.2. For every extraction, an additional sum and product node is added to the network. The child of the product node that represents the approximately rank-one sub-matrix (atom) can now be described with a leaf distribution. Only the two remaining datasets, as indicated in Figure 2.2 by subset 1 and subset 2, still need to be further expanded.

By iteratively extracting these leaf distributions, the data gets compressed until all the extracted data matrices are close to rank-one. The process is then completed and a resulting network is generated. This SPN-SVD algorithm achieved higher accuracies than the LearnSPN algorithm on 20 binary and discrete generative datasets (Adel *et al.*, 2015). The SPN-SVD algorithm was also extended to work on discriminative datasets such as the MNIST dataset. At the time of publication, the authors were able to achieve a state-of-the-art discriminative classification accuracy for SPNs of 97.6% using Discriminative SPN-SVD (DSPN-SVD), which is a discriminative version of the SPN-SVD algorithm. In a discriminative setting, one wants to predict which class in the output variable is the most likely given the input data provided. The DSPN-SVD algorithm first finds a subset of the input variables that are maximally correlated with each class of the output random variable. The SPN-SVD algorithm is then called for each class using the most correlated subset of input variables for that class. Thus for every class label, an SPN is constructed over the input random variables found to be the most correlated to that class.

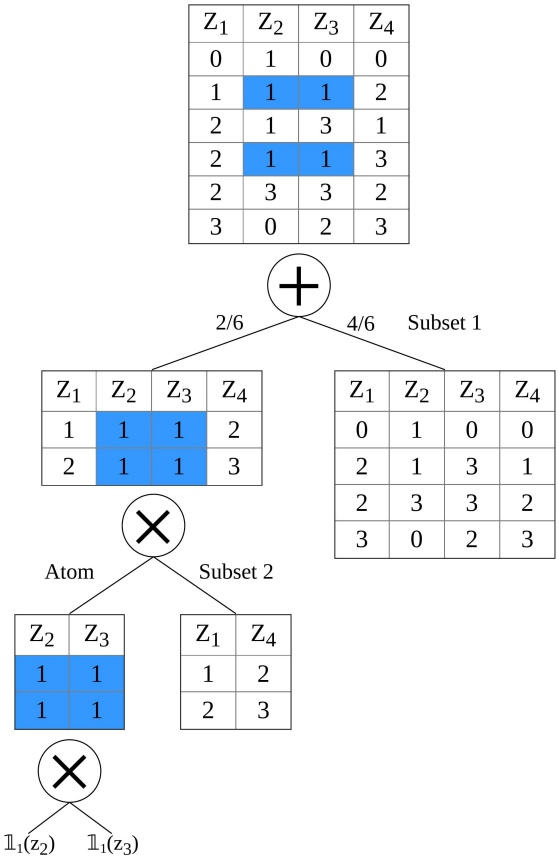



*Figure 2.2: Illustration of how the SPN-SVD expands a network using a sub-matrix extraction. The distribution every node represents is indicated above that node. Note that the data atom (indicated in blue) is converted into a leaf distribution. The indicator function, $\mathbb{1}_k(z_i)$, is a leaf distribution and is mathematically described in Section 3.3.*

#### 2.3.2.3 MIXCLONES algorithm

Both LearnSPN and SPN-SVD rely on constructing the complete SPN structure from data. Once the network has been constructed, it cannot easily be expanded on using the same learning algorithm. Consequently, if more time is provided to these learning algorithms, they cannot increase the likelihood of the data given the network any further. A new anytime structure learning algorithm was proposed by Dennis & Ventura (2015) called the MIXCLONES structure learning algorithm. This MIXCLONES algorithm is called an anytime algorithm as it can be stopped at any reasonable time in its execution

and it will provide a valid SPN that approximately fits the data. The longer the algorithm runs, the better the network usually describes the data. This algorithm works by starting with a naive initial SPN that assumes all the random variables are independent in the network. The algorithm then expands parts of the network that seem to describe the data the least accurately. Additional nodes are then added to these parts to make them slightly better at describing the data.

The MIXCLONES algorithm iteratively executes 3 steps until the network becomes sufficiently accurate to be useful. The first step is to identify the correct product node to expand. A product node represents the assumption that all its children are independent from one another. The algorithm wants to find which of these independence assumptions describes the data the worst. The goal of structure learning is to find the network that maximises the likelihood of the data given the network and the prior probabilities over networks. The algorithm therefore tries to find the product node that least contributes to the likelihood score. This product node is then selected for expansion. The second step of the algorithm is then to find the two children of the product node that have the most correlation with each other. These two children nodes are therefore considered to not be as independent as the network assumes. The algorithm then clones the product node and its two most correlated children, as shown in Figure 2.3. In the third step, the weights are then also updated to fit the data better. The network should now have a slightly higher likelihood. Due to some approximations made in identifying the weakest product node, it is possible that the likelihood of the network did not increase after the expansion is made. In that case, the expansion is undone and the product node and children combination is added to a blacklist. This blacklist is a temporary list that helps the learning algorithm to avoid making the same expansion mistakes again.

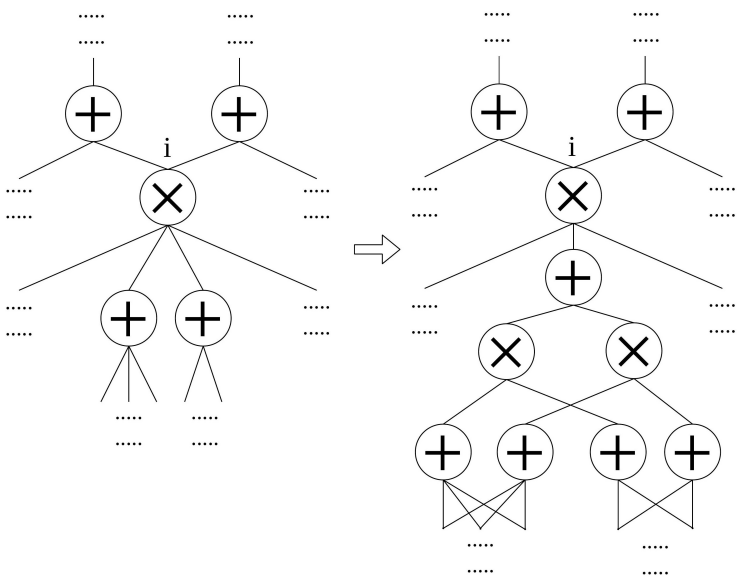



*Figure 2.3: Illustration of how the MIXCLONES algorithm expands a product node into two clones of that product node. The weight values of the clones are also updated, and this allows the network to slightly better represent the data.*

#### 2.3.2.4 Prometheus algorithm

The Prometheus algorithm (Jaini *et al.*, 2018) was proposed as a structure learning algorithm that also constructs a complete SPN from data. It thus works similarly to LearnSPN and SPN-SVD. As with LearnSPN and SPN-SVD, Prometheus splits the data as it constructs a network in a top-down fashion until it reaches leaf distributions. Prometheus also alternates between data clustering, represented by a sum node, and variable partitioning, represented by product nodes, as the LearnSPN algorithm does. The main difference between Prometheus and the other two fixed-learning algorithms is that it creates multiple variable partitions of the data at every partition step. This means that it does not only partition the data once but creates multiple possible random variable partitions, represented by multiple product nodes. A sum node is then added and signifies a mixture distribution over these partitioned product nodes. This multiple partitioning of a data table, instead of just one, increases the correlation the network can capture between random variables.

An obvious problem with having multiple partitions at every time step is that the

network can become large very quickly. To combat this problem, Prometheus combines all substructures that are similar to one another in structure and weight configuration. This decreases Prometheus's memory requirements while also reducing redundant calculations.

### 2.3.3 Network compression

Every node in an SPN represents a distribution over a subset of random variables. After an SPN is constructed there might be distributions that are similar and can be merged. Compression algorithms for SPNs try to reduce as much redundant structure in a network as possible without sacrificing accuracy. In Rahman & Gogate (2016) a compression algorithm was proposed that tests every node's distribution against every other similar node's distribution. If two distributions are found to be close enough to one other, they can be merged to form one distribution. This merging reduces the network's parameters while not sacrificing any significant accuracy.

Another compression algorithm, proposed in Ko *et al.* (2018), works by converting a valid SPN into a tensor version of an SPN, called tensor-SPN (tSPN). An SPN is therefore effectively converted into a tensor network. Tensor networks are networks composed of mathematical representations called tensors. The reason this conversion is done is because there are effective methods of compressing tensor networks. If an SPN is converted into a tSPN, it can, therefore, be significantly compressed, upwards of 90% parameter compression, with negligible loss in accuracy. However, the compressed version of the SPN is usually not a valid SPN any longer, but still a valid probability distribution. This is due to the tSPN only trying to reduce the computational requirements as much as possible, and does not require it to be a valid SPN. Due to a tSPN usually not being a valid SPN it is challenging to predict what weights and structures can be updated, while still ensuring the network remains a valid probability distribution. This means that once an SPN is converted into a tSPN, no large-scale changes can easily be made to the network without risking the network losing its probabilistic properties.

### 2.3.4 Partial observability and explainability

SPNs, as with other probabilistic models, have a wide range of interpretability properties, which can be exploited. However, in SPNs, there has been no known specific research conducted on generating an explanation for a prediction made by the model. Research

has, however, been done on working with partially observed inputs. In Poon & Domingos (2011), the authors used SPNs for face completion. The network took in partially observed inputs and was tasked with predicting missing pixels in the image of a human face. The SPN would, for example, get pixels for the left half of the image of a human face, and then be tasked with predicting the right half. They relied on a method called approximate MPE inference, which is discussed in Section 4.5. Using generative gradient descent they were able to achieve state-of-the-art performance at the time of publishing.

## 2.4 Conclusion

Probabilistic models are extremely useful when one wants a model that can provide answers to a wide range of questions around the variables being modelled. This is in contrast to deep neural networks, which are some of the most accurate models in existence but struggle with flexible inference. Furthermore, probabilistic models usually suffer from either not being accurate enough or struggling with providing answers in a reasonable time-frame. One possible solution to this problem is the use of SPNs, which can represent a wide range of data types to a high degree of accuracy while also having fast inference times.

While SPNs have the promise of providing the best of both worlds for probabilistic models (being fast and accurate), in practice they remain hard to train due to their strict structural requirements. In SPN literature we see a range of learning algorithms being developed for SPNs, which fall under parameter learning and structure learning algorithms. There are also two methods of training on data, namely the discriminative and generative methods. Lastly, these algorithms can also work with two types of variables, namely discrete and continuous variables. We summarise the capabilities of each learning algorithm in Table 2.1.

*Table 2.1: Overview of learning algorithms for SPNs presented in this literature study. Note that in the training setting category, 'Both' means that the algorithm can work for both discriminative and generative training. In the datatype category, the word 'Both' means that the algorithm can work in both the discrete and continuous domains.*

| Algorithm | Anytime | Learn structure | Training setting | Datatype |
|---|---|---|---|---|
| Gradient descent | Yes | No | Both | Both |
| Expectation Maximisation | Yes | No | Generative | Both |
| LearnSPN | No | Yes | Generative | Discrete |
| (D)SPN-SVD | No | Yes | Both | Both |
| MIXCLONES | Yes | Yes | Generative | Both |
| Prometheus | No | Yes | Generative | Both |

It would be of interest to investigate and implement a learning algorithm that can more easily be applied to a wide range of training configurations as indicated by the categories in Table 2.1. In previous research, no learning algorithm was found that could satisfy every condition in Table 2.1 and therefore we try to design such an algorithm in this work. There is also hype around SPNs being more interpretable than other non-probabilistic algorithms. It would therefore be interesting to evaluate which properties of an SPN can be used to make them more interpretable. We start by first describing the theory behind SPNs and why they have these attractive properties, as described in the next section.

# Chapter 3

# Theoretical background

## 3.1 Probability theory

Probabilistic models such as SPNs work from a set of rules defined by probability theory. Probability theory is a branch of mathematics that is used to analyse and model uncertainty. Two fundamental rules of probability theory are called the sum and product rules. These two rules further lead to another important rule, called Bayes' rule or Bayes' theorem. This section serves as an introduction to the next section on probabilistic models.

### 3.1.1 Sum rule

The sum rule states that a variable can be marginalised out of a probability distribution by summing over every possible state of that random variable. This is described by

$$p(z_1) = \sum_{\forall z_2} p(z_1, z_2), \tag{3.1}$$

where $p(z_1)$ indicates a marginal probability distribution over random variable $z_1$. Similarly, $p(z_1, z_2)$ indicates a joint probability distribution over random variables $z_1$ and $z_2$. The random variable $z_2$ has a discrete number of states in equation (3.1). If $z_2$ is a continuous random variable it can be marginalised out using,

$$p(z_1) = \int_{-\infty}^{\infty} p(z_1, z_2) dz_2. \tag{3.2}$$

If $z_2$ has a fixed state-value range it is defined over, the integral is taken over that particular range.

### 3.1.2 Product rule

The product rule states that a joint probability distribution can be split into a marginal distribution multiplied by a conditional distribution, and is described by

$$p(z_1, z_2) = p(z_1|z_2)p(z_2) = p(z_2|z_1)p(z_1). \tag{3.3}$$

If $z_1$ and $z_2$ are independent from one another, equation (3.3) becomes the product of two marginal distributions

$$p(z_1, z_2) = p(z_1|z_2)p(z_2) = p(z_1)p(z_2). \tag{3.4}$$

### 3.1.3 Bayes' rule

We can now use equation (3.2) and equation (3.3) to form Bayes' rule, which is defined as

$$p(z_1|z_2) = \frac{p(z_2|z_1)p(z_1)}{p(z_2)} = \frac{p(z_2|z_1)p(z_1)}{\int_{-\infty}^{\infty} p(z_1, z_2)dz_1}. \tag{3.5}$$

Alternatively if $z_1$ is a discrete random variable Bayes' rule becomes

$$p(z_1|z_2) = \frac{p(z_2|z_1)p(z_1)}{p(z_2)} = \frac{p(z_2|z_1)p(z_1)}{\sum_{\forall z_1} p(z_1, z_2)}. \tag{3.6}$$

## 3.2 Training probabilistic models

Let us now consider two sets of random variables. The first set of random variables are input random variables, represented with the vector $\mathbf{x}$. The second set of random variables are the output random variables, for which we want to infer probabilities, and are represented with the vector $\mathbf{y}$. The goal of a probabilistic model is now to model some distribution over these sets of random variables ($\mathbf{x}$, $\mathbf{y}$). We would like a model to predict $p(\mathbf{y}|\mathbf{x})$ to be as close to the underlying distribution, which the data is generated from, as possible. The random variables that are in the input and output sets can also be chosen at testing time, depending on what probabilistic query one wants to ask. Therefore, different

types of probabilistic queries can be asked by changing which variables are inputs and which are outputs. However, depending on how the model is trained, some probabilistic queries might be better modelled than others. There are generally two ways of training a probabilistic model, namely the generative and discriminative training setting. In the discriminative setting, the model tries to directly model $p(\mathbf{y}|\mathbf{x})$, for some set of fixed input and output random variables. If one would move random variables from $\mathbf{x}$ to $\mathbf{y}$, at testing time, the accuracy of the model might diminish. This is because the probabilistic model was not trained to represent a distribution over these moved random variables which were in $\mathbf{x}$ at training time. The discriminative setting is useful when one knows that some random variables will always be in set $\mathbf{x}$ at testing time and therefore the model does not need to represent $p(\mathbf{x})$ as well. Discriminative training is used in e.g. image classification, where the input random variables (pixels) are always in set $\mathbf{x}$ (or marginalised out in the case of partial inputs). Therefore, a distribution over these pixels does not need to be modelled. In the classical discriminative case we, therefore, might have one class random variable in the set $\mathbf{y}$ and all the other input feature random variables in the $\mathbf{x}$ set.

If one does not want to specify what random variables will be outputs ($\mathbf{y}$) and inputs ($\mathbf{x}$) at training time, the generative training setting might be more appropriate. In the generative setting we directly model both $p(\mathbf{x}|\mathbf{y})$ and $p(\mathbf{y})$, which is equivalent to modelling $p(\mathbf{x}, \mathbf{y})$. This is due to

$$p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x}|\mathbf{y})p(\mathbf{y}). \tag{3.7}$$

Therefore, from now on we refer to the generative setting as modelling $p(\mathbf{x}, \mathbf{y})$, as we will later see that SPNs directly output $p(\mathbf{x}, \mathbf{y})$ in one network forward pass. The output value, $p(\mathbf{y}|\mathbf{x})$, we are interested in can now be calculated for the generative setting. To do this we use Bayes' rule as follows

$$p(\mathbf{y}|\mathbf{x}) = \frac{p(\mathbf{x}, \mathbf{y})}{p(\mathbf{x})} = \frac{p(\mathbf{x}, \mathbf{y})}{\int_{-\infty}^{\infty} p(\mathbf{x}, \mathbf{y}) d\mathbf{y}}. \tag{3.8}$$

Generative training is useful in cases such as image completion, when one set of random variables (input pixels) are given and another set of random variables (output pixels) should be inferred. We therefore would still like our model to represent the probabilistic query $p(\mathbf{y}|\mathbf{x})$ as accurate as possible, but now we do not specify beforehand which random variables will be in $\mathbf{y}$ and $\mathbf{x}$ at testing time. As in the case with image completion, different

pixels can be used as input and output at testing time. It is therefore necessary to model a joint distribution over all the random variables, as is done in the generative setting. In the generative setting it does not matter which random variables are in the sets $\mathbf{x}$ and $\mathbf{y}$ at training time as a joint probability over all the random variables are modelled.

We now assume a probabilistic model is trained from data points. Each data point is described over two subsets, the data for the output ($\mathbf{y}$) set, denoted by $\mathbf{y}_n$, and data for the input ($\mathbf{x}$) set, denoted by $\mathbf{x}_n$, for a specific data point at index $n$. The complete dataset for the output ($\mathbf{y}$) random variables is represented by $\mathbf{y}_{1:N}$, and all the data for the input random variables ($\mathbf{x}$) is represented by $\mathbf{x}_{1:N}$. The integer $N$ represents the number of data points. We now define the discrete random variable $m$ to represent different model structures. This can be different models such as e.g. Gaussian or Exponential models, or different types of SPN structures as they can model different distributions. We therefore assume there is a discrete set of models to choose from. The parameters of each of these models are represented with the vector $\boldsymbol{\theta}_m$. The variables in $\boldsymbol{\theta}_m$ can be a combination of discrete or continuous values. We now want to find an appropriate model and its parameters to best fit our data.

We first look at the case where the model is provided and only the parameters need to be estimated from data. We would therefore like to obtain

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, p_m(\boldsymbol{\theta}_m|\mathbf{y}_{1:N}, \mathbf{x}_{1:N}) = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, \frac{p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\boldsymbol{\theta}_m)p_m(\boldsymbol{\theta}_m)}{p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N})}, \tag{3.9}$$

where $\hat{\boldsymbol{\theta}}_m$ represents the best parameters for model $m$ to fit the data. The probability distribution $p_m$ represents a probability distribution over the model parameters and training data for the given model $m$. Because $p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N})$ is not dependent on $\boldsymbol{\theta}_m$ it remains constant for changes in $\boldsymbol{\theta}_m$. This term therefore has no effect on the optimisation process and can be removed. The new parameter optimisation goal is therefore represented by

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\boldsymbol{\theta}_m)p_m(\boldsymbol{\theta}_m). \tag{3.10}$$

Equation (3.10) tries to best represent the joint probability of the data. This is what we want in the case of generative learning. For discriminative learning we do not want to explicitly model $p_m(\mathbf{x}_{1:N}|\boldsymbol{\theta}_m)$ and therefore, using equation (3.3), the discriminative parameter optimisation goal becomes

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, p_m(\mathbf{y}_{1:N}|\mathbf{x}_{1:N}, \boldsymbol{\theta}_m)p_m(\boldsymbol{\theta}_m). \tag{3.11}$$

Alternatively to parameter learning, in model selection (or structure learning), we might have different models to chose from. We would, therefore, like to select a model structure that optimally represents our data. In this case we assume that for every model we already have a set of parameters. We would therefore like to find

$$\hat{m} = \underset{m}{\operatorname{argmax}}\, p(m|\mathbf{y}_{1:N}, \mathbf{x}_{1:N}) = \underset{m}{\operatorname{argmax}}\, \frac{p(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|m)p(m)}{p(\mathbf{y}_{1:N}, \mathbf{x}_{1:N})}, \tag{3.12}$$

where $\hat{m}$ represents the best model that fits the data. The probability term $p(\mathbf{y}_{1:N}, \mathbf{x}_{1:N})$ is not dependent on $m$ and can therefore be discarded out of the optimisation process. In the case of selecting a model we would therefore like to optimise

$$\hat{m} = \underset{m}{\operatorname{argmax}}\, p(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|m)p(m) = \underset{m}{\operatorname{argmax}}\, p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\boldsymbol{\theta}_m)p(m), \tag{3.13}$$

where $\boldsymbol{\theta}_m$ is the parameters chosen for that model $m$. Note that if $\boldsymbol{\theta}_m = \hat{\boldsymbol{\theta}}_m$, for every model $m$, equation (3.13) would provide the optimal model and parameters to represent the data distribution. Again, equation (3.13) tries to model the joint probability over all the random variables, as is done in the generative setting. In the discriminative version of equation (3.13) we do not want to explicitly model $p(\mathbf{x}_{1:N}|m)$, otherwise written as $p_m(\mathbf{x}_{1:N}|\boldsymbol{\theta}_m)$. Therefore in the discriminative setting we want to maximise

$$\hat{m} = \underset{m}{\operatorname{argmax}}\, p(\mathbf{y}_{1:N}|\mathbf{x}_{1:N}, m)p(m) = \underset{m}{\operatorname{argmax}}\, p_m(\mathbf{y}_{1:N}|\mathbf{x}_{1:N}, \boldsymbol{\theta}_m)p(m). \tag{3.14}$$

We now want to represent both the discriminative and generative optimisation goals using the same mathematical expression and therefore reduce the need to write duplicate mathematical terms. To do so we define a normalising function as

$$Z(p_o, g) = \begin{cases} 1, & \text{if } g = \text{True} \\ p_o, & \text{if } g = \text{False} \end{cases}, \tag{3.15}$$

where $g$ is a boolean variable that represents whether the model should be trained generatively or not. If the model should be trained generative $g$ = True and if it should be trained in the discriminative setting $g$ = False. The term $p_o$, which can be a value, vector or matrix, represents the normalising of probability output(s) in the discriminative training setting.

Parameter learning for a specific model can now be represented with

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, \frac{p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\boldsymbol{\theta}_m)p_m(\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_{1:N}|\boldsymbol{\theta}_m), g)}. \tag{3.16}$$

Note that, in the generative setting, we are maximising the model to best represent the joint probability distribution over the data. In the discriminative setting, we are maximising the model to best represent the conditional probability distribution over the data. The reason we write our optimisation goal as a fraction, with joint and marginal probabilities, is because SPNs only compute joint or marginal probability values. Therefore, this fractional form becomes important in later sections where SPNs are trained. By assuming that each data point is generated independently from one another, we then get the total likelihood for parameter learning to be maximised as

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}} \frac{p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\boldsymbol{\theta}_m)p_m(\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_{1:N}|\boldsymbol{\theta}_m), g)} = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, p_m(\boldsymbol{\theta}_m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)}. \quad (3.17)$$

Similarly, we can also rewrite the model selection process in the generative and discriminative settings to be

$$\hat{m} = \underset{m}{\operatorname{argmax}} \frac{p_m(\mathbf{y}_{1:N}, \mathbf{x}_{1:N}|\boldsymbol{\theta}_m)p(m)}{Z(p_m(\mathbf{x}_{1:N}|\boldsymbol{\theta}_m), g)} = \underset{m}{\operatorname{argmax}}\, p(m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)}. \quad (3.18)$$

Therefore, by maximising equation (3.17) and (3.18) we can find probabilistic models that fit our data well. The prior probability terms over possible models ($p(m)$) and possible parameters for those models ($p_m(\boldsymbol{\theta}_m)$) can help to reduce overfitting and training times on a given dataset if chosen right. We use $p_m(\boldsymbol{\theta}_m)$ in parameter learning to reduce the training times by controlling how large each weight can become. In structure learning, we use the $p(m)$ prior to help combat against overfitting.

We would also like the capabilities to train a probabilistic model to be partially generative and partially discriminative, as is needed in Section 4.7.2.2. We sometimes want a network to have relatively good discriminative classification accuracies, while also having slight generative properties. To do this we simply adapt equation (3.17) to form

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, p_m(\boldsymbol{\theta}_m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{p_m(\mathbf{x}_n|\boldsymbol{\theta}_m)^{1-p_g}}, \quad (3.19)$$

where $p_g$ is a value in the range $[0, 1]$, which indicates how generatively the network should be trained. If $p_g = 1$ the network is trained in the completely generative setting and if $p_g = 0$ the network is trained in the completely discriminative setting. The value

$p_g$ can, however, also be any value between zero and one. Similarly, we can also rewrite the model selection process to be

$$\hat{m} = \underset{m}{\operatorname{argmax}}\, p(m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{p_m(\mathbf{x}_n|\boldsymbol{\theta}_m)^{1-p_g}}. \tag{3.20}$$

The probabilistic model we investigate in this work is the sum product network. Before continuing we first need to define what an SPN is, as is described in the next section.

## 3.3 Understanding sum product networks

To understand what an SPN is, it is useful to consider an example of a generative SPN. A generative SPN is used for illustrative purposes, but this explanation can be extended to the discriminative case. In this example, we use discrete random variables, although SPNs can also model continuous random variables. In the generative setting, the goal of an SPN is simply to find a compact network that accurately represents a joint probability distribution. The main reason why one wants to find a compact form to represent this joint probability distribution is to allow the network to generalise well and therefore provide better results on new unseen data. A second reason why one wants to compress a distribution is to reduce the memory and computational requirements of doing probabilistic inference.

An example of a joint probability distribution over two random variables is presented in Table 3.1.

*Table 3.1: An example of a joint probability distribution, where $z_1$ and $z_2$ are discrete random variables, having respectively 2 and 3 possible states.*

| $z_1$ | $z_2$ | $p(z_1, z_2)$ |
|---|---|---|
| 0 | 0 | 0.06 |
| 0 | 1 | 0.09 |
| 0 | 2 | 0.00 |
| 1 | 0 | 0.14 |
| 1 | 1 | 0.21 |
| 1 | 2 | 0.50 |

We would like to find a more compact representation of this data. At this point we are not concerned with generalisations but with compactly representing this distribution. A simple example of how compression can lead to better generalisation is provided in Section 5.1. It is important to note that even though these two random variables are not independent, there is conditional independence in these distributions, which can be exploited. Exploiting conditional independence is an important property that makes SPNs able to represent a wide range of distributions while remaining tractable.

Before we express Table 3.1 in equation form, we first define the indicator function $\mathbb{1}_k(z_j)$, over a discrete random variable $(z_j)$, as

$$\mathbb{1}_k(z_j) = \begin{cases} 1 & z_j = k \\ 0 & z_j \neq k \end{cases}. \tag{3.21}$$

Therefore, if the discrete random variable is at state $k$ the indicator function outputs a one, or otherwise a zero. The indicator function can be viewed as representing a discrete probability distribution over a discrete random variable, where the distribution outputs a non-zero probability value for only one state of that random variable $(z_j = k)$. The joint probability in Table 3.1 can now be represented as an SPN in equation form as

$$\begin{aligned} p(z_1, z_2) &= 0.06\mathbb{1}_0(z_1)\mathbb{1}_0(z_2) + 0.09\mathbb{1}_0(z_1)\mathbb{1}_1(z_2) + 0.14\mathbb{1}_1(z_1)\mathbb{1}_0(z_2) \\ &\quad + 0.21\mathbb{1}_1(z_1)\mathbb{1}_1(z_2) + 0.5\mathbb{1}_1(z_1)\mathbb{1}_2(z_2) \\ &= 0.5[0.3\mathbb{1}_0(z_1) + 0.7\mathbb{1}_1(z_0)][0.4\mathbb{1}_0(z_2) + 0.6\mathbb{1}_1(z_2)] + 0.5\mathbb{1}_1(z_1)\mathbb{1}_2(z_2). \end{aligned} \tag{3.22}$$

The zero probability term in Table 3.1 is discarded as the probabilistic network only needs to represent variable configurations that are possible. By exploiting conditional independence, the joint probability distribution can be factorised as shown in the last step in equation (3.22).

One can now do a joint probabilistic query by setting the random variables to specific values and in the process activating specific indicator variables, using equation (3.22). Marginalising out random variables is also possible by setting all the indicators for the random variables one wants to marginalise out to 1, as is further discussed in Section 4.4. Any conditional probability distribution can also be calculated using Bayes' rule, therefore using equation (3.22) twice, e.g.

$$p(z_2|z_1) = \frac{p(z_1, z_2)}{p(z_1)}, \tag{3.23}$$

where $p(z_1)$ indicates that $z_2$ has been marginalised out. Therefore any joint, marginal or conditional probabilistic query can be performed depending on how the network's inputs are assigned. The whole joint probability table in Table 3.1 can now be represented as an SPN, as illustrated in Figure 3.1.

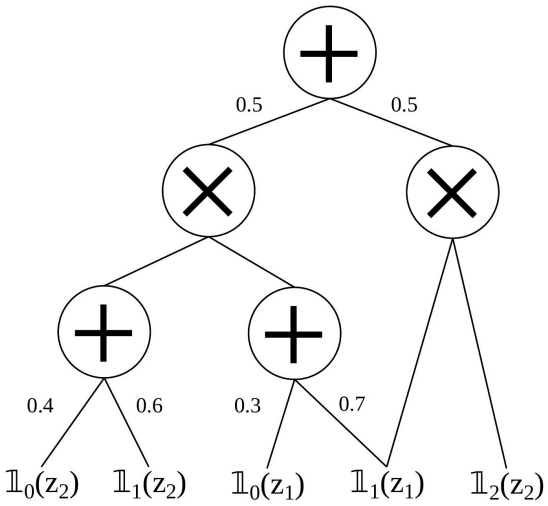



*Figure 3.1: Equivalent SPN generated from the joint probability distribution in Table 3.1.*

In Figure 3.1 the product nodes represent local conditional independence assumptions between random variables. The variables do not need to be completely independent to exploit independence between them. The sum nodes can be seen as mixture models since they sum up their weighted inputs to produce an output. If we do not count the zero probability term, the factorised SPN has slightly more parameters than the original joint distribution. This is due to the small size of the joint probability table. For a larger joint probability table, an SPN typically uses fewer parameters than the original table, as we later see in some of the results in Section 7. There we even train networks of sizes smaller than 30 000 parameters on a discrete input version of the MNIST dataset. The complete joint probability for this dataset would have been 60 000 parameters (e.g. number of data points). We now investigate the calculations needed to perform one pass through the unfactorised and factorised networks, both described in equation (3.22). For simplicity, we are not counting indicator functions as computations. In this case, a computation is defined as any multiplication or summation between two values. The factorised SPN uses 11 calculation steps instead of the 14 that would have been needed with the unfactorised

version (excluding the zero probability term). This is only a small improvement due to the small size of the joint probability table. For larger tables, however, factorising a distribution uses considerably less space and computation time than the unfactorised version.

## 3.4 Defining sum product networks

### 3.4.1 Nodes of an SPN

Sum product networks represent a probability distribution over random variables. Once an SPN is constructed, any joint, marginal or conditional probability query can be answered in time linear to the size of the network. An SPN consists internally of sum and product nodes with univariate or multivariate leaf distributions. These networks recursively build on smaller valid SPNs. Every node in an SPN is a valid SPN and is either a leaf distribution, e.g. Bernoulli or Gaussian, or an internal node defined by

$$p_i(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) = \begin{cases} \sum_{j \in \text{children}(i)} w_{ij} p_j(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}) & i \text{ is a sum node} \\ \prod_{j \in \text{children}(i)} p_j(\mathbf{x}^{(j)}, \mathbf{y}^{(j)}) & i \text{ is a product node} \end{cases}, \tag{3.24}$$

where node $j$ represents one of the children of node $i$. Here $p_i(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$ and $p_j(\mathbf{x}^{(j)}, \mathbf{y}^{(j)})$ are the outputs of nodes $i$ and $j$ over subsets, $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}$ and $\{\mathbf{x}^{(j)}, \mathbf{y}^{(j)}\}$, of the complete random variable set $\{\mathbf{x}, \mathbf{y}\}$. Notice that the children of a sum node need to be defined over the same random variables as the sum node itself. The weight connecting parent sum node $i$ and its child node $j$ is defined as $w_{ij}$ with a constraint of $w_{ij} > 0$. For discrete random variables, leaf distributions are normally univariate Bernoulli distributions. However, sum nodes with more than two indicator function children, defined over the same discrete random variable, can also be used as a leaf distribution. For leaf distributions over continuous random variables the univariate Gaussian distribution is normally used. Each sum node can be thought of as creating a mixture model of child distributions, where the weighting $w_{ij}$ specifies the importance of each mixture component. A product node can be thought of as representing a joint distribution over a set of assumed independent child distributions. The root node or top node of the network represents the output of the network in the form of a joint or marginal probability value. This probability value is

calculated by setting all the random variables to specific values and performing a forward pass through the network.

### 3.4.2 Validity of an SPN

Before we proceed to define what a valid SPN is, we first need to introduce two supporting concepts. The first is that every node of the SPN, in combination with its children, is also an SPN. Our definition of validity will, therefore, have to apply to all nodes in the SPN. Secondly, we need to define the 'scope' of a node. The scope of a node is defined as the union of all the random variables it operates on (i.e. represents a distribution of). With that in place, we now turn our attention to the two requirements of a valid SPN (Kalra *et al.*, 2018):

- Completeness: For sum nodes, every child node has to have the same scope as its siblings, i.e. all siblings operate on the same set of random variables.
- Decomposability: For product nodes, every child node must have a scope different from that of its siblings, i.e. all siblings operate on disjoint sets of random variables. This means that a specific random variable cannot be present in more than one of the children of a product node.

Therefore if the network is complete and decomposable, it is considered to be valid. If an SPN is valid, with sum node weights greater than zero and constructed from valid probabilistic leaf distributions, it is a probabilistic distribution. This means that the output value presented at the root node is a valid output of an unnormalised probability distribution. If one wants the SPN to present normalised probability distributions, an additional constraint needs to be imposed. This constraint states that every sum node's weights should sum to one.

## 3.5 Conclusion

Probabilistic models are a class of machine learning algorithms that explicitly model uncertainty. These models can provide a wide range of probabilistic answers without the need to be trained on each possible query configuration. SPNs are a specific kind of probabilistic model that can compactly represent a joint probability distribution by

making use of conditional independence between random variables over subsets of data. Therefore, two random variables do not need to be completely independent for SPNs to be able to compactly represent them. This allows SPNs to compactly represent large joint probability distributions while also having fast inference times. SPNs have strict structural requirements to remain a valid probability distribution, which can complicate learning these models from data. Due to SPNs having linear inference time in the size of the network, they have some interesting and useful capabilities. In the next chapter, we discuss what some of these capabilities are, and we create a new algorithm to potentially help better interpret their predictions.

# Chapter 4

# Capabilities of sum product networks

In this chapter we provide an overview of the different inference and query capabilities of SPNs. We will assume that an SPN has already been trained from data, as described in Chapters 5 and 6.

## 4.1 Joint and marginal inference

As described in Section 3.3, SPNs can be seen as a factorised form of a joint probability table with efficient inference capabilities. Any probability value in this joint probability table can be retrieved in time linear to the size of the network. A bonus property of SPNs is that any marginal probability value, i.e. a probability value with only a subset of the total random variables specified, can also be retrieved in time linear to the size of the network. These joint and marginal properties allow any conditional probability value to be retrievable using only two passes through the network. This conditional value can be retrieved by using

$$p(\mathbf{z}_1|\mathbf{z}_2) = \frac{p(\mathbf{z}_1, \mathbf{z}_2)}{p(\mathbf{z}_2)}, \tag{4.1}$$

where $\mathbf{z}_2$ represents the set of random variables that are observed and $\mathbf{z}_1$ represents the set of random variables one wants to do inference on. One problem that might arise with probability values is that they may become extremely small when a high number

of random variables are being modelled. The next section therefore describes how to practically represent these small probabilities in an SPN.

## 4.2 Handling low probability values

When working with a large number of random variables, e.g. image data, probability values can become extremely small. Let us consider a binary $28 \times 28$ input pixel image, which equates to 784 input random variables. If we naively assume all images are equally likely, we get a probability of each image to be $0.5^{784}$, which is an extremely small decimal number, approximately equal to $10^{-236}$. Probability values become even smaller when one works with a high number of continuous random variables. In these cases, the probability density outputs of a network regularly produce values of size $10^{-1600}$. These values are smaller than the smallest positive real number a 64-bit floating-point number can represent at full precision. Therefore images and other data with a high random variable count can cause numerical underflow in these 64-bit floating point numbers, and thus the network can incorrectly output a zero probability value. We also cannot simply use a larger number representation as it decreases training speeds and increases inference times. To combat this problem, all SPN operations are conducted in the logarithmic domain. If we convert $10^{-1600}$ to the logarithmic domain with base $e$ it becomes $-3684.1\ldots$, which can easily be represented as a 64- or even 32-bit float. The logarithmic domain is also perfectly suited for probability values, due to probabilities being strictly positive. The logarithmic function maps values in the range of $(0, 1]$ to $(-\infty, 0]$ in the logarithmic domain. Working with a wider range of values in a 64-bit float thus helps to represent the real probability more accurately.

To work in the logarithmic domain we simply convert all leaf-distribution outputs and indicator-function outputs to their logarithmic equivalent values, between the range of $(-\infty, 0]$. The rest of the internal (sum and product) nodes in the network now directly work with these logarithmic values. To represent the real probability value of 0 in the logarithmic domain, an infinity large negative number is needed. This value can, however, be approximated in the logarithmic domain by simply using a large negative number like $-10^{10}$, with negligible loss in accuracy. The sum and product operations now need to be performed directly in the logarithmic domain. A product over probability values can be

calculated in the logarithmic domain using

$$\ln(p_1 \times \cdots \times p_K) = \ln(p_1) + \cdots + \ln(p_K), \tag{4.2}$$

where $p_1$ to $p_K$ represent all the child probability values one wants to multiply together. Here $K$ represents the number of children of the product node. The natural logarithm (log with base $e$) is represented by the ln operator. The natural logarithm is used for simplicity of implementation as it simplifies gradient calculations. These gradient calculations are important for fast multi-configuration inference and gradient descent parameter learning, as is discussed in Sections 4.6 and 5.2.1.

The second operation we would like to perform is summation. Summation requires a few extra steps to compute reliably in the logarithmic probability domain. A mathematical trick, called the Log Sum Exponential (LSE) trick, is used to avoid ever needing to represent the real probability as a number, and therefore avoid underflow. We actually do perform summation in the linear (real) probability domain, but we first subtract each number by a common value in the logarithmic domain to avoid underflow. The complete summation can be performed using

$$\ln(w_1 p_1 + \cdots + w_K p_K) = \ln[e^{\ln(w_1)+\ln(p_1)-M} + \cdots + e^{\ln(w_K)+\ln(p_K)-M}] + M, \tag{4.3}$$

where $K$ is again the number of child probability values of that sum node. The sum node's weight values associated with every child are represented by $w_1, \ldots, w_K$. The value $M$ can be computed using

$$M = \max_{k \in [1,K]} (\ln(w_k) + \ln(p_k)), \tag{4.4}$$

and represents the common factor we want to remove from each child probability value. Thus equation (4.4) focusses on maintaining precision on the highest value in the summation. Any precision loss will only occur if values are extremely small compared to the highest weighted probability value, and at that point they have a negligible effect on the summation result in any case. All computations in an SPN can now be performed in the log probability domain without ever needing to represent the real probability values. The output of the entire SPN will now produce the log probability of the input query.

## 4.3 Inference with continuous and discrete random variables

Up until this point, we have only demonstrated how SPNs work for discrete random variables. SPNs can, however, also work with continuous random variables. SPNs can effectively be constructed from any tractable leaf probabilistic distribution, over a subset of the random variables associated with each SPN. In this example, we will focus on the univariate Gaussian distribution, as it can easily be implemented while still being quite expressive. This example can, however, be extended to other distributions which might also be multivariate.

The univariate Gaussian density function is defined as

$$f(z|\mu,\sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(z-\mu)^2}{2\sigma^2}}, \tag{4.5}$$

where $z$ represents the continuous state value of the continuous random variable being modelled. The values $\mu$ and $\sigma$ represent the mean and standard deviation of the Gaussian distribution. As stated in Section 4.2, SPN calculations are performed in the logarithmic domain to avoid numerical errors. Therefore by converting equation (4.5) to its logarithmic form we derive

$$\ln(f(z|\mu,\sigma)) = -\frac{1}{2}\ln(2\pi) - \ln(\sigma) - \frac{(z-\mu)^2}{2\sigma^2}. \tag{4.6}$$

We can now use this univariate Gaussian density distribution in an SPN. If one works with discrete random variables, the indicator function is used. The indicator function, defined in equation (3.21), can now be written in its logarithmic form as

$$\mathbb{1}_k(z_j) = \begin{cases} 0 & z_j = k \\ -\infty & z_j \neq k \end{cases}, \tag{4.7}$$

where $z_j$ is the random variable the indicator function is defined over, and $k$ is the state value that the indicator function turns on for. The $-\infty$ can practically be represented as a large negative number, e.g. $-10^{10}$, with negligible error in the network's output. A simple SPN configuration with a discrete random variable, $z_1$, and continuous random variable, $z_2$, is illustrated in Figure 4.1.

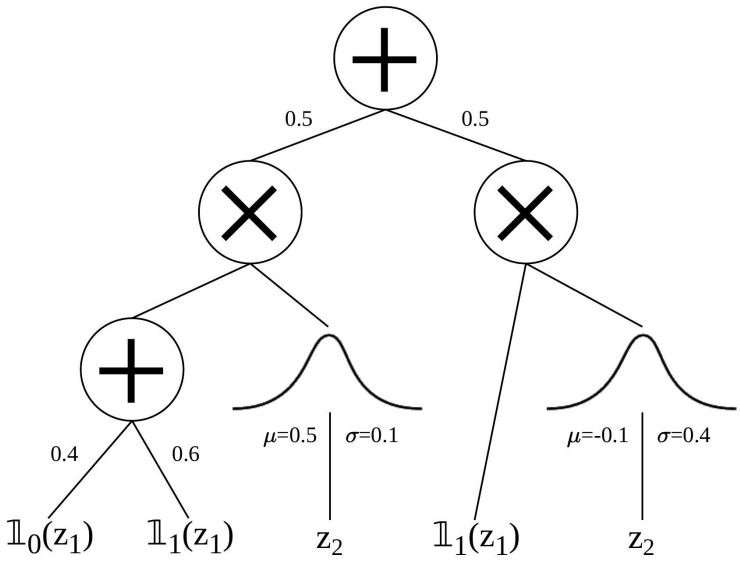



*Figure 4.1: An example SPN defined over a discrete random variable, $z_1$, and continuous random variable, $z_2$. Two univariate Gaussians leaf distributions are used for the continuous random variable $z_2$. In this example, $z_2$ has a state-space of $(-\infty, \infty)$. The mean and standard deviations of each Gaussian are written underneath each Gaussian.*

In Figure 4.1 there are effectively 4 leaf distributions. The two continuous leaf distributions are represented by the two univariate Gaussians. The discrete leaf distributions are simple Bernoulli distributions over the random variable $z_1$. The first Bernoulli distribution is represented by the bottom left sum node in Figure 4.1. For the second Bernoulli distribution, only one state of $z_1$ has a non-zero probability and can, therefore, be described by a single indicator function. This indicator function is seen in the bottom-right of Figure 4.1. The SPN in Figure 4.1 can now be evaluated as normal by calculating values from the bottom of the network upwards. When a Gaussian node is encountered, one simply uses equation (4.6) to calculate the output of the node.

It is important to note that if the outputs of leaf distributions defined over continuous random variables in a network are probability density values, the SPN also outputs a probability density value. Otherwise, if one wants to determine the probability of each continuous random variable being between two state values, i.e. a probability estimate, the network outputs a probability value.

## 4.4 Partially observed inputs

Unlike some other probabilistic networks such as PGMs, SPNs have fast and exact inference on marginalised inputs. This means that an SPN can provide an exact probability estimation in time linear to the size of the network, even if some of the random variables are not observed. This is a useful property to have in real-world applications where all the input random variables are not always known. If a discrete random variable is unobserved one can simply set all the corresponding indicator functions defined over that variable to one (zero in the logarithmic domain). This effectively marginalises that random variable out. Similarly, a continuous random variable can be marginalised out by simply setting the output of each leaf distribution, defined over that random variable, to one (or zero in the logarithmic domain). Let us again consider the probability distribution shown in Table 4.1.

*Table 4.1: Example joint probability, where $z_1$ and $z_2$ are discrete random variables, having respectively 2 and 3 possible states.*

| $z_1$ | $z_2$ | $p(z_1, z_2)$ |
|---|---|---|
| 0 | 0 | 0.06 |
| 0 | 1 | 0.09 |
| 0 | 2 | 0.00 |
| 1 | 0 | 0.14 |
| 1 | 1 | 0.21 |
| 1 | 2 | 0.50 |

We would now like to calculate the marginal probability distribution

$$p(z_2 = 1) = \sum_{\forall z_1} p(z_1, z_2 = 1), \tag{4.8}$$

where $z_1$ and $z_2$ are discrete variables, having respectively 2 and 3 possible states. To do this we first convert the probability distribution in Table 4.1 to an equivalent SPN. The probability value $p(z_2 = 1)$ can now be calculated in one pass through the network by setting all the indicator functions associated with $z_1$ to 1 and $\mathbb{1}_1(z_2) = 1$, as illustrated in Figure 4.2.

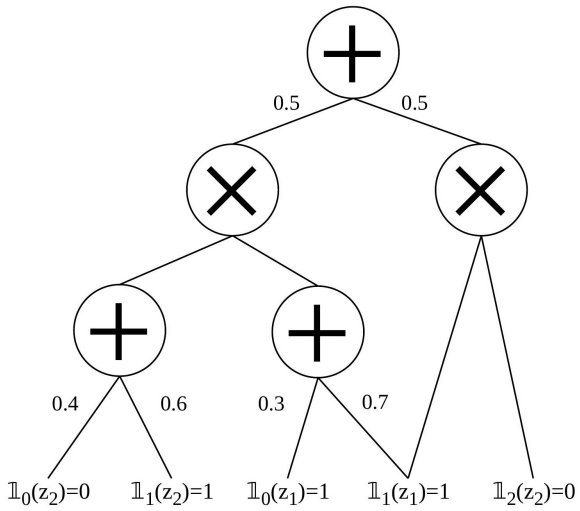



*Figure 4.2: Example SPN illustrating the marginal probability calculation of* $p(z_2 = 1)$. *Note that all indicator functions associated with* $z_1$ *are set to* 1.

Therefore, all joint and marginal probability queries, over the set of random variables the SPN is defined over, can be calculated by performing one pass through the network. By doing a forward pass through the network we get $p(z_2 = 1) = 0.3$. We can now use this marginalisation property to predict the most likely state of unobserved random variables as well. How this is done is described in the next section.

## 4.5 Approximate most probable explanation

Sum product networks can also provide an approximate Most Probable Explanation (MPE) to given input data by doing two passes through the network. Let us assume that an SPN is defined over two sets of random variables. MPE refers to the network's ability to predict, given one set of observed random variables, what the most probable states of the other set of random variables would be. It is important to note that any number of random variables can be observed and the MPE algorithm will estimate the most likely state for all the other random variables. This capability of SPNs is used extensively for image completion where one half of an image is given and the network needs to predict what the other half looks like (Gens & Domingos, 2012).

Although SPNs cannot guarantee to find the true MPE of an observation in time linear to the size of the network, the approximate MPE can be found in time linear to the network size. This approximation has been shown to achieve state-of-the-art results in many variable completion challenges (Van de Wolfshaar & Pronobis, 2019). This approximate MPE can be achieved by first setting all the random variables that are observed to their corresponding values. The indicator values for discrete random variables, which we want to predict the most likely state for, are all set to the value one (zero in the logarithmic domain). We set all outputs of continuous leaf distributions, defined over random variables we would like to infer the most likely state of, to one (zero in the logarithmic domain), as we would have done for marginalisation. We then conduct a forward pass through the network with the replacement of all sum nodes with max nodes. A max node takes the maximum value of its weighted inputs and produces this value as its output. The output of the network is now the probability of the approximately most likely states of the random variables. Lastly, we backtrack through the network by selecting for every max node the child with the highest weighted probability value. For product nodes, we select all its children nodes. This process is continued until we reach an indicator function or a continuous leaf distribution. For an indicator function, the state that the function turns on for is the most likely state of the discrete random variable it is defined over. For a continuous leaf distribution one simply takes the maximum density value of that distribution as the state value for the random variable it is defined over.

An illustration of how the MPE process works is shown in Figure 4.3. In Figure 4.3, $z_1$ represents the observed random variable, and $z_2$ the random variable we would like to infer about. We would thus like to infer the most likely value of $z_2$ given that we know $z_1$ is in state 1. The left image in Figure 4.3 represents the forward pass calculated through the network with a final probability of the approximate most likely state being 0.336. We can now backtrack from the root node to find the most likely variable configuration. This process can be seen in the right image in Figure 4.3. The most likely state is thus $z_1$ in state 1, as specified, and $z_2$ in state 1. We can, therefore, perform an approximate MPE in time linear to the size of the network.

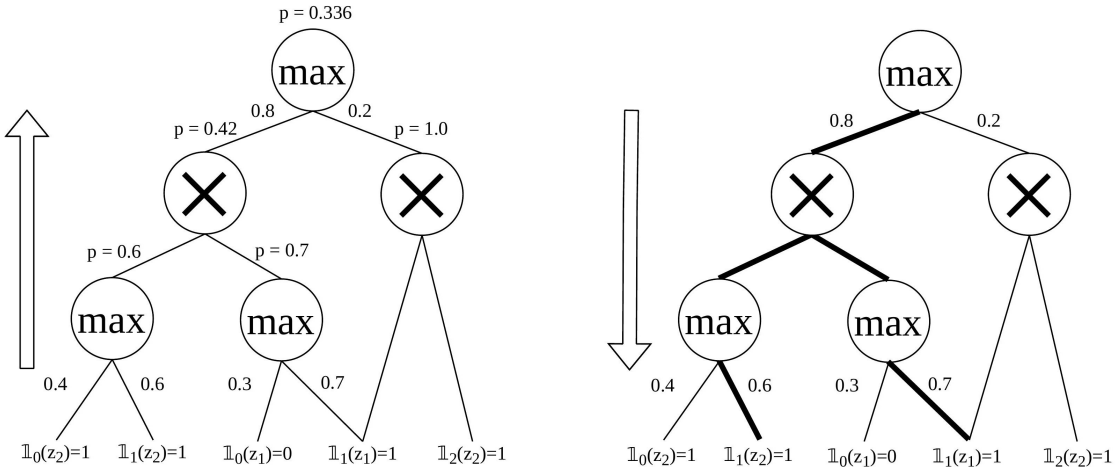



*Figure 4.3: Example of performing approximate MPE inference in an SPN by doing a forward pass through the network followed by a backward pass. In this case, the observed random variable is $z_1$, with state value 1. The random variable we want to infer the state of is $z_2$. Note the selection path, illustrated with a dark black line, indicating that the approximate most likely state value estimated for $z_2$ is 1. The root max node selects the first child product node because it has a weighted output of $(0.6 * 0.7)0.8 = 0.336$ compared to the weighted output of $0.2$ of the other child product node.*

SPNs can also infer more than one probability value by performing only two passes through the network. This is an interesting property that involves using gradients in the network as is discussed in the next section.

## 4.6 Fast multi-configuration inference

Sum product networks have a lesser-known ability to infer the probability of multiple random variable configurations while needing only four passes through the network. The values that need to be calculated are $p(\mathbf{y}, \mathbf{x})$, and $p(\mathbf{x})$ in the discriminative setting, as well as the gradients of both outputs with respect to each leaf distribution. As no normalisation is needed in the generative training setting, for a normalised SPN, only one forward pass ($p(\mathbf{y}, \mathbf{x})$) and one backward gradient pass is needed through the network to apply this multiple-inference technique. We discovered that SPNs have this ability through the investigation of a closely related set of probabilistic networks named Arith-

metic Circuits (AC). These ACs can also be trained from data (Lowd & Domingos, 2012), but have slightly stricter structural requirements. To our knowledge, this is the first time that multi-configuration inference has been adapted and used in SPNs.

In SPNs, as with ACs, it is possible to cheaply re-evaluate the network's output for any query that differs from the original query by only one random variable having a different state value. For example, let us consider an SPN defined over two discrete random variables, $z_1$ and $z_2$, with three and two states respectively. We now perform the query $p(z_1 = 2, z_2 = 0)$ on this SPN. By using gradient values we can exactly find the probabilities $p(z_1 = 0, z_2 = 0)$, $p(z_1 = 1, z_2 = 0)$, and $p(z_1 = 2, z_2 = 1)$, without needing to re-evaluate the network in each case. Note that all these probabilistic queries only have one random variable that differs from the original query. This is possible in SPNs because there is a linear relationship between the output of the network and any one given leaf distribution's output, which is further discussed later in this section. A linear function has an interesting property that the gradient value associated with this function remains constant for any input value to this function. Therefore, if we calculate the gradient values for the output of the network with respect to each leaf distribution, we can exactly predict the network's output for a change in any one leaf distribution, without needing to re-evaluate the SPN. We can, however, only accurately predict the network's output for changes in leaf distributions defined over the same random variables. This means that we can use the gradient values calculated using the original query to predict the network's output for changes in one random variable, by only re-evaluating the new leaf distribution outputs over that random variable. If, for example, we have calculated the probability of a specific image, we can now also infer the network's output for any one-pixel change from the original image. This may appear to be an insignificant change, but we show that this property is useful for generating explanations, as can be seen in Sections 4.7.2.

To achieve this multiple-query capability, we first set all the network's random variables to specific values corresponding to the original probabilistic query being asked. We then perform a forward pass through the network. The next step is to calculate the rate of change of the network's output with respect to each indicator function ($\mathbb{1}_k(z_i)$), represented by $g_k(\sim z_i)$, and continuous-leaf distribution, represented by $\frac{\partial p_r}{\partial p_l}$. The symbol $p_r$ represents the output of the root node and $p_l$ represents the output of a leaf node at

index $l$. The value $g_k(\sim z_i)$ is calculated in the same manner as calculating the gradient value for the output of the network with respect to a continuous leaf distribution, over the random variable $z_i$, if it was located where the indicator function is. Therefore, we calculate both these gradient values in exactly the same manner and it is only the notation that differs. We use the notation $g_k(\sim z_i)$, because the indicator function is a discrete function and therefore we cannot directly use the partial derivative notation. The $\sim z_i$ part of the expression indicates that this gradient value is dependent on all the random variables except random variable $z_i$, which implies that the gradient value remains constant for changes in only $z_i$. We show how to calculate gradient values in time linear to the size of the network in Section 5.2.1.1.

To illustrate how this multi-configuration inference is possible, we again look at an example joint probability distribution. This probability distribution is presented in Table 3.1 and 4.1 and is written in equation form as

$$\begin{aligned} p_r = {} & 0.06\mathbb{1}_0(z_1)\mathbb{1}_0(z_2) + 0.09\mathbb{1}_0(z_1)\mathbb{1}_1(z_2) + 0.14\mathbb{1}_1(z_1)\mathbb{1}_0(z_2) \\ & + 0.21\mathbb{1}_1(z_1)\mathbb{1}_1(z_2) + 0.5\mathbb{1}_1(z_1)\mathbb{1}_2(z_2), \end{aligned} \tag{4.9}$$

where $z_1$ and $z_2$ has two and three states, respectively. The value $p_r$ represents the output probability of the network (root node). Equation (4.9), therefore, represents the computation an equivalent SPN would perform for the given joint probability table. We can now factorise equation (4.9) with respect to any of the random variables. In this example we factorise equation (4.9) with respect to $z_2$ and get

$$\begin{aligned} p_r = {} & \mathbb{1}_0(z_2)[0.06\mathbb{1}_0(z_1) + 0.14\mathbb{1}_1(z_1)] + \mathbb{1}_1(z_2)[0.09\mathbb{1}_0(z_1) + 0.21\mathbb{1}_1(z_1)] \\ & + \mathbb{1}_2(z_2)[0.5\mathbb{1}_1(z_1)] \\ = {} & \mathbb{1}_0(z_2)g_0(z_1) + \mathbb{1}_1(z_2)g_1(z_1) + \mathbb{1}_2(z_2)g_2(z_1). \end{aligned} \tag{4.10}$$

In equation (4.10) we see that the output of the network can be written in terms of indicator functions over one random variable, and gradient values where $g_k(\sim z_2) = g_k(z_1)$. An interesting property of SPNs is that these gradient values are always defined over variables other than the selected random variable with regards to which we factorised. This is because a gradient value, of the network's output with respect to a leaf distribution, is equal to the sum of all the values multiplied with that leaf distribution in the forward pass before reaching the output node. As described in Section 3.4, in SPNs only weight values and probability outputs of disjoint (different) random variables can

be multiplied together. Therefore, this gradient value can only be composed of random variable outputs different than the one the leaf distribution is defined over. For changes in only the selected random variable (e.g. $z_2$), these gradient values remain constant. This means that the gradient values only need to be calculated once, and from then only the leaf distributions over $z_2$ need to be recalculated for changes in $z_2$. This same rule applies for continuous random variables, where instead of discrete leaf distributions we have continuous leaf distributions. The gradient values of the output with respect to every leaf distribution can also easily be calculated in time linear to the size of the network, as shown in Section 5.2.1.1.

General equations for calculating the output of a network where only one random variable's state is changed can now be created for discrete and continuous random variables. For a discrete random variable this can be done using

$$p_r(z_i^*) = \sum_k \mathbb{1}_k(z_i^*) g_k(\sim z_i), \tag{4.11}$$

where $z_i^*$ represents the new state value for the discrete random variable $z_i$, which is the random variable with the changed state value. Here $\mathbb{1}_k(z_i^*)$ is the new output of one the indicator functions, at index $k$, defined over the discrete random variable $z_i$. The gradient value $g_k(\sim z_i)$ represents the gradient value calculated using the original query. The value $p_r(z_i^*)$ represents the predicted, but accurate, output of the network for this new probability query, with the changed state value. Therefore, by summing over all the random variable's indicator functions, multiplied by their gradients, a new output can be predicted. For a continuous random variable the output of the network can be predicted using

$$p_r(z_i^*) = \sum_l p_l(z_i^*) \frac{\partial p_r(z_i)}{\partial p_l(z_i)}, \tag{4.12}$$

where $p_l(z_i^*)$ is the new output of one of the leaf distribution at index $l$, defined over the continuous random variable $z_i$. The gradient value $\frac{\partial p_r(z_i)}{\partial p_l(z_i)}$ represents the gradient value calculated using the original query.

While an SPN normally operates in the logarithmic domain, we illustrate this SPN property in the linear probability domain. This multiple-inference property, however, works just as well in the logarithmic domain. Figure 4.4 illustrates the forward and backward pass performed through the network to calculate the gradient values. In this

example, we work with two discrete random variables, $z_1$ and $z_2$, with state values 0 and 1 respectively.

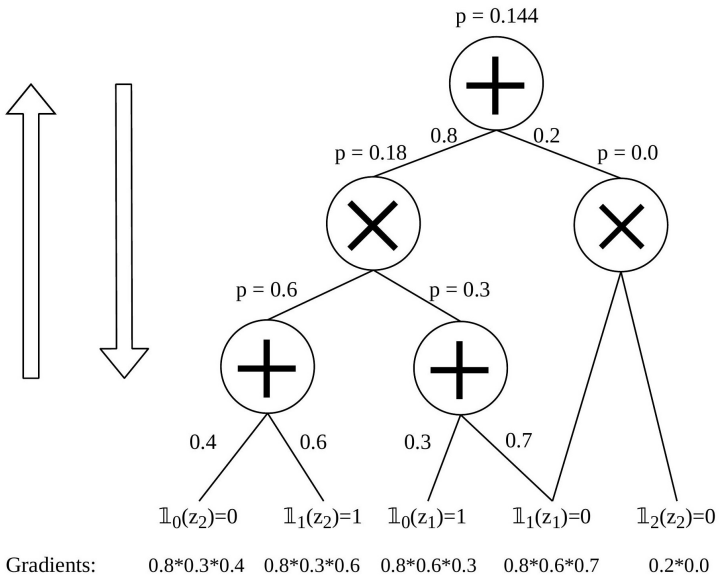



*Figure 4.4: An SPN setup where one forward pass and one backward pass is conducted through the network. In the forward pass the p values, indicating the probability output of each node, are calculated. In the backward pass the gradients with respect to each indicator function ($g_k(\sim z_i)$) are calculated using backpropagation, as further described in Section 5.2.1.1. Note that the gradients of the two indicator functions, with output values of 1, are equal to the output of the network, as is expected.*

In Figure 4.4 one can see that the output of the network, given $z_1$ changes to state 1, would be $0.8 \times 0.6 \times 0.7 = 0.336$. The probability $p(z_2 = 1)$, therefore marginalising out $z_1$, becomes $0.8 \times 0.6 \times 0.3 + 0.8 \times 0.6 \times 0.7 = 0.48$.

This is an extremely useful tool for SPNs with random variables with a high number of states. A practical application of this technique is in training discriminative SPNs with an output variable with a large number of labels. A probability value for every one of those labels can therefore be calculated by doing at most two passes (or four in the discriminative setting) through the network. We will also exploit this property to make generating explanations for predictions easier, as shown in Section 4.7.2.

## 4.7 Interpretability

### 4.7.1 Generating new data

Another interesting property of SPNs is that they can create new data points that are closely related to the data they were trained on. Generating new data samples is a useful method of determining what features a network is focussing on, especially in the discriminative setting. In this discriminative setting, for example, the SPN's sampling ability helps a person estimate what features the network thinks are important to each class. Therefore, by inspecting the important features one can see if the network is indeed focussing on the right properties for each class. As a side note, sampling SPNs becomes even more accurate when combined with an autoencoder, which is a neural network architecture that finds lower-dimensional representations of data, to clean up the data being generated (Dennis & Ventura, 2017a).

To sample from a normalised SPN is a relatively simple process. Sampling can be performed by simply recursively moving downwards, starting from the root node, by choosing child nodes until a leaf distribution is selected. For every sum node that is encountered, we select one child node at random. The probability of selecting a child node is represented by the weight of the sum node for that particular child node. Therefore, if a child node has a weighting of e.g. 0.5, the probability of selecting that child node is 0.5, given the SPN is normalised. If an SPN is not already nominalised it can simply be normalised first before using this sampling ability. For a product node, the paths through all its children are chosen. In both cases (product and sum node selection), if one arrives at an indicator function, the state that the indicator function outputs one for is the state selected for the discrete random variable it is defined over. If a continuous leaf distribution is encountered, a random weighted sampling of the random variable's state can be calculated. The leaf distribution's density function therefore acts as the probability of each state value for that continuous random variable. An example of this data sampling in an SPN is shown in Figure 4.5, where the sampled data point is $z_1 = 0$ and $z_2 \approx 0.6$.

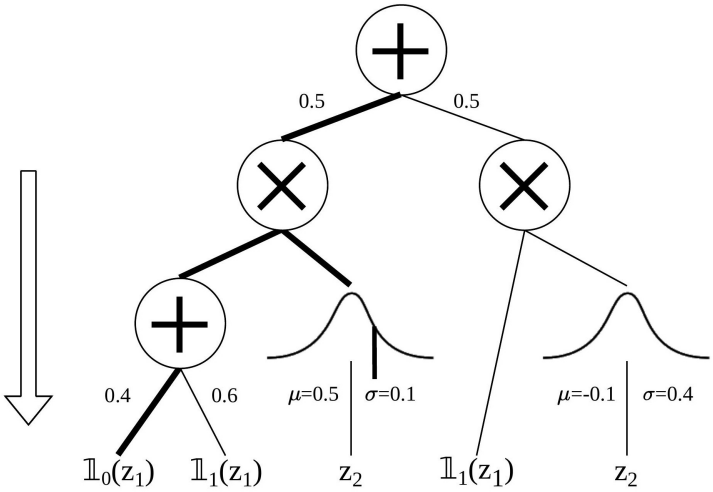



*Figure 4.5: Random sampling (indicated with the dark black lines) using an SPN to create new data. Note that the largest weighted child is not always chosen, as a large weight only means the probability of that child being chosen is high.*

### 4.7.2 Explainability

Due to SPNs having certain probabilistic properties, they also have the potential to partially explain why they are making a certain prediction. We try to show this by deriving a simple algorithm for doing so, called Explain-SPN, by exploiting the fast-inference property described in Section 4.6. This algorithm is intended to attempt to provide a fast and rudimentary explanation for why a network is making a certain prediction.

#### 4.7.2.1 The Explain-SPN algorithm

Explainability is an important property for models to have when critical decisions need to be made using these models. The model therefore does not only provide a prediction to a given input, but can also be leveraged to indicate why it has made this prediction. We will explain and test, in Section 4.7.2.2, how this algorithm works on image data, although this algorithm can also theoretically be used on other types of data. The goal of the Explain-SPN algorithm, in the case of images, is to take as input an example image and a class. We would then like the Explain-SPN algorithm to provide and basic explanation as to why the given image might represent this given class. To do this, the Explain-SPN

algorithm generates an image which is a slight alteration from the original image. This altered image should slightly better represent the given class, according to the trained network. The network is therefore used to generate an image that exaggerates some characteristics of the given class using the original image. These characteristics might be hard to see in the original image and therefore exaggerating them might allow a user to better spot them in the new image and trace this back to the original image. It is also possible that the Explain-SPN algorithm generates invalid explanation images, as we also explore in Section 4.7.2.2. An observer is then able to use this new image to determine if they have missed some important features in the original image or whether the network has generated an invalid explanation. We will now explore how these explanation images are generated.

Once an SPN has been trained in the discriminative setting it can be used to classify new data. Thus, given some inputs, the network can provide a probability distribution over the output classes, which corresponds to what it predicts that inputs represent. If we now want to know approximately what the network is basing its predictions on, we can attempt to find how important or unimportant (independent) each input random variable is to the network's output prediction. If an input random variable is approximately independent in regards to the prediction of the network's output, given the other input random variables, it is true that

$$p(y|x_1, \ldots, x_i, \ldots, x_d) \simeq p(y|x_1 \ldots, x_{i-1}, x_{i+1}, \ldots, x_d), \tag{4.13}$$

where $x_1, \ldots, x_i, \ldots, x_d$ represent all the input random variables. The value $i$ represents the specific input random variable index one wants to marginalise out. The value $d$ represents the number of input random variables. Marginalisation therefore tells the network to disregard any information from this input in making its prediction. If the output of the network remains approximately the same, it means the prediction is made approximately independent of this input, given the other inputs.

In the first step of the Explain-SPN algorithm, we marginalise out each input random variable, one at a time, and see what effect each random variable has on the network's output. This process of marginalising out random variables is slow if performed in the usual manner of two forward passes, for discriminative classification, per marginalisation. However, as described in Section 4.6, SPNs can calculate a range of probability outputs by conducting only four passes through a network in the discriminative setting. This is

achieved by first performing two forward passes (joint and marginal) on the current data point. Two backward passes are then conducted to calculate gradient values which are used to predict how the network's output would change for any single variable change. Thus the network's output for marginalising each input random variable individually can be predicted using these gradients. This drastically speeds up the Explain-SPN algorithm's execution time.

We would now like to generate a rudimentary explanation image using the input image and the specified class. We can change each pixel's greyscale value in accordance with how much marginalising changes the output of the network. If the accuracy stays relatively the same after marginalising, for a certain random variable, this pixel value is set to grey which indicates it is approximately independent of the output. We could also just have used the pixel value of the original image instead of grey for the explanation image, but decided to use grey as we can then also see which pixels are important to the network's output prediction. If the accuracy goes up (above a certain threshold) after marginalisation, it means the assignment of this value is serving the prediction negatively. Therefore, the pixel colour is simply set to the value it is furthest from, e.g. white if it was more black in the original image. The thought process behind this colour change is that we naively predict that the network wants this pixel to be around the opposite value it currently is. If the network accuracy goes down when this pixel is marginalised, it means the assignment of this pixel value is serving the prediction positively, and the colour is set to the closest value of zero (white) or one (black) according to the value of the original image. The generated explanation image should now be slightly more fit to represent the given class, while also not differing too much from the original image. This greyscale image version of the Explain-SPN algorithm is further described in Algorithm 1.

**Algorithm 1** Explain-SPN algorithm used to generate an explanatory image. Note that each pixel has a value in the range of $[0, 1]$. Here 0.5 represents the colour grey and the round function simply rounds the value to either 0 (white) or 1 (black) corresponding to which number is closest.

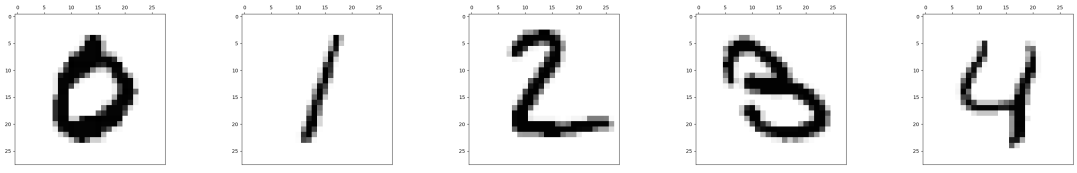
**Data:** Input oldImg, cutOff.
**Result:** Explanation image newImg.
probOut = Output of forward passes using oldImg.
Do backward passes and calculate gradients.
Initialise new greyscale image array newImg.
**for** $i$ in number of input random variables **do**
  Predict probMarg by marginalisation out variable[$i$].
  probChange = probMarg-probOut
  **if** *probChange* > *cutOff* **then**
    newImg[$i$] = round(1.0-oldImg[$i$])
  **else if** *probChange*<-*cutOff* **then**
    newImg[$i$] = round(oldImg[$i$])
  **else**
    newImg[$i$] = 0.5
  **end**
**end**

Algorithm 1 can be used to generate an image that partially explains why the network is recommending a certain class. We, therefore, believe that with more research better explainability algorithms can be created to allow SPNs to be more interpretable. We now evaluate the potential of this method to generate explanations.

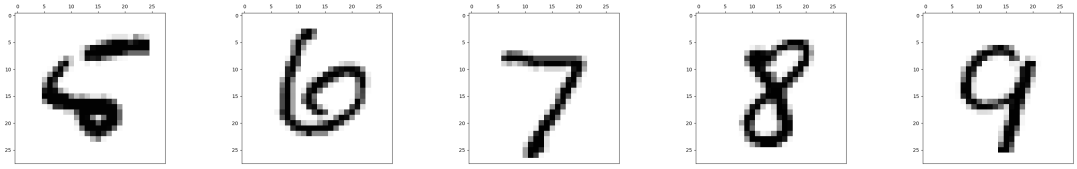
#### 4.7.2.2 Investigation on Explain-SPN

In Section 4.7.2.1, we derived an algorithm for generating basic, human interpretable, explanations to the predictions the network makes. We would now like to provide an example of how the output of the Explain-SPN looks on a test image in the MNIST dataset. Each data point in this dataset represents a two-dimensional image, composed of $28 \times 28$ greyscale values, and a corresponding digit label between the values of 0 and 9. The network is tasked with classifying these digits for every input image. These 10 types of digits in the MNIST dataset are represented in Table 4.6. The MNIST dataset

is also described in more detail in Section 7.1.1 and includes a reference on where to find this dataset.



*(a) Zero* *(b) One* *(c) Two* *(d) Three* *(e) Four*



*(f) Five* *(g) Six* *(h) Seven* *(i) Eight* *(j) Nine*

*Figure 4.6: Examples of the 10 different classes/digits present in the MNIST dataset.*

For this experiment, an SPN is trained using the structure learning algorithm we derived, called SET-SPN. The inner workings of the SET-SPN algorithm are described in Chapter 6. In this training setting we used 25% of the training data as validation data. The validation dataset is used to determine when to stop training a network. If the likelihood of a network starts to decrease on the validation dataset it serves as an indication that training should stop, as the network is probably starting to overfit on the training data. Overfitting, described in Section 5.1, means that the network is starting to model noise in the training data, which deteriorates the model's accuracy on testing data. Therefore, the network is trained using the remaining 75% of the MNIST training set. We also train the SPN to be partially generative and partially discriminative using equation (3.19), where $p_g$, with ranges $[0, 1]$, again indicates how generatively the network should be trained. The optimiser is set up to focus mainly on discriminative training ($p_g = 0.2$). An epoch has passed when the learning algorithm has successfully updated the network's weights using each of the data points exactly once. The reason we train the network to be slightly generative is because we would like the network to be able to partially explain why it has made a certain prediction in addition to making the prediction itself. We start by first training the network in this discriminative and partially generative manner and achieve a raw classification accuracy of 97.1%. This is a slightly lower classification

accuracy than a purely discriminatively trained SET-SPN, but this network has an added capability of partially being able to explain itself, as is described next.

We now investigate the capabilities of this SPN to partially explain why it is making a certain prediction. Figure 4.7 shows the original digit that was selectively chosen, due to it possibly representing an 8 or 9.

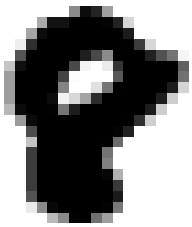


*Figure 4.7: A selectively chosen MNIST image representing the digit 8. This image is chosen as it could possibly also have represented the digit 9.*

The trained network estimates the probability of it being an 8 to be 91%. It estimates an 8.5% probability of it being a 9 and a 0.4% probability of being a 6. The other digits make up the remaining 0.1% of the network's predictions. We now apply the Explain-SPN algorithm, presented in Section 4.7.2.1, to generate an image that supports its claim of being an 8. The algorithm does this by colouring all pixels that do not change the network output by more than 0.5% to be grey. If marginalising the pixel decreases the network's prediction of it being an 8 by more than 0.5% it means that this pixel's current value is important to its claim. The pixel value is therefore rounded to the closest absolute value white (0) or black (1). If marginalising the pixel increases the network's prediction by more than 0.5% it means that this pixel's current value does not support the network's prediction. This pixel is therefore inverted and rounded to the closest value of white (0) or black (1). This essentially means that we estimate the network wants this pixel to be the opposite colour to better support its claim. This new pixel value is what we predict the network would like to have seen here. The resulting image can be seen in Figure 4.8.

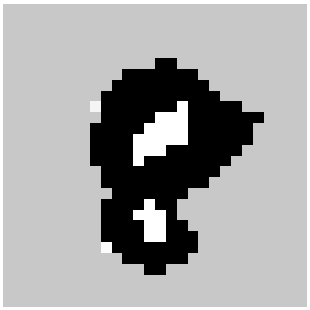
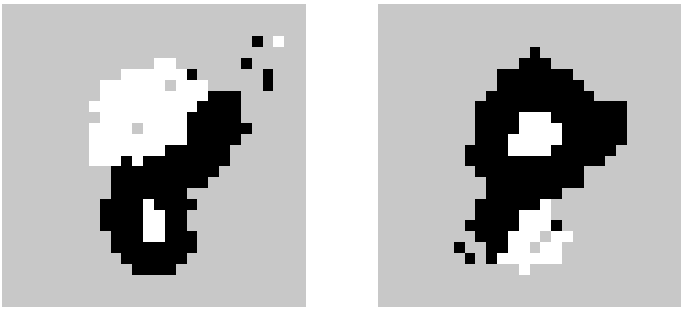


*Figure 4.8: The tri-coloured version of the image in Figure 4.7. The network produced this image after being prompted to generate an image that is slightly more likely to represent the digit 8. Note the newly formed whitened area in the middle of the eight's bottom rounding.*

As can be seen in Figure 4.8, the SPN does a fairly decent job at generating a slightly more likely image to represent the digit 8. It even captures and exaggerates the whitened bottom rounding that is usually present in a normal instance of an 8 digit. A user can now use this explanatory image to see that the network thinks the bottom black area of the original image should correspond to the bottom rounding of an 8. Therefore, the SPN can provide a prediction as well as provide a partial explanation of why it made a certain prediction. We now investigate if the algorithm can also explain other classes, using the same image, even though the class was not the most likely prediction. Figure 4.9 represents the results generated by the Explain-SPN algorithm. In Figure 4.9 (a) the network is tasked with explaining why the original image might be a 6. In Figure 4.9 (b) the network indicates why it might be a 9. The network seems to be able to find fairly decent explanation images that support both claims, even though the explanation images have incorrect exaggerations in them. A person can now use these images as evidence that the network has made an incorrect prediction as it is classifying certain parts of the image incorrectly. There is also more noise present in the image than in Figure 4.8 as there are more pixels the network needs to change. Even though the network does not predict it to be a 6 or 9, it can still provide partial evidence for both.



*(a) Explanation of digit 6* *(b) Explanation of digit 9*

*Figure 4.9: Here the Explain-SPN algorithms is used to try and explain why it might be a 6 and 9 being represented in the original image (Figure 4.7). Note that in image (a) the network inverted the black pixels in the top left corner as this makes a 6 more likely. In image (b) it is the bottom left right pixels that were inverted.*

These results are also not bound to image data. Image data was used as it can easily be visualised and interpreted. Although there were some images that this method could not generate a clear explanation for, it does provide a proof of concept to the idea of explainability in SPNs. With some additional research, we believe that a more advanced algorithm would be able to generate significantly better explanations to the predictions the network makes.

## 4.8 Conclusion

In this chapter, we investigated the different inference capabilities that SPNs have. We found that they can answer any joint, marginal and conditional probability query in time linear to the size of the network. We also found that SPNs can find an approximate, most probable explanation in time linear to the size of the network. We then investigated a unique property of SPNs of being able to provide multiple, but selective, probability outputs without the need to do multiple passes through the network. This property was used to design a simple algorithm, called Explain-SPN, which can generate explanations of why the network is making a certain prediction.

We ended off by experimenting with the Explain-SPN algorithm to help generate explanations to its predictions. We found that the Explain-SPN could generate, in time

linear to the size of the network, human interpretable images that can help a person understand what pixels the network is focussing on. The network was then tasked with generating an explanation of it being a certain specified digit, even though the network did not predict this class to be the most likely class. It was still able to approximately generate an image that shows how that digit might possibly have been represented in the image.

SPNs have some additional interesting properties due to their probabilistic nature, such as generating new data that looks similar to the data it was trained on. This can provide a useful tool to analyse what features a network is focussing on.

To allow for these inference capabilities to be useful, the network must first accurately model data provided to it. To do this the model must be trained using that data. In the next chapter, we describe the mathematics behind different parameter learning algorithms to help us train SPNs from data.

# Chapter 5

# Parameter learning in sum product networks

To get a desired SPN that models a certain distribution, it has to be trained from data from that distribution first. Sum product networks can generally be trained from data using two different techniques, namely parameter learning and structure learning. Parameter learning involves first defining a valid starting structure, usually with randomly assigned weights, and then adjusting the weights of that network. Structure learning attempts to simultaneously learn the structure and weights of a network directly from data. Structure learning is generally considered more difficult as it poses this extra challenge of also learning the structure. In this chapter we provide an overview of how learning works in SPNs as well as evaluate different parameter learning algorithms for SPNs.

## 5.1 Objective of learning

### 5.1.1 Viewing generalising as compression

It is important to remind oneself why one goes through the exercise of trying to find a model that describes certain data. The reason models are used is to attempt to generalise to new unseen examples while also reducing inference and computational times. We therefore want a model to take existing data and compress it into a lower-dimensional form, which we assume would generalise better to new real world data. This compression of data is what gives a model better predictive accuracies. We will now show how SPNs

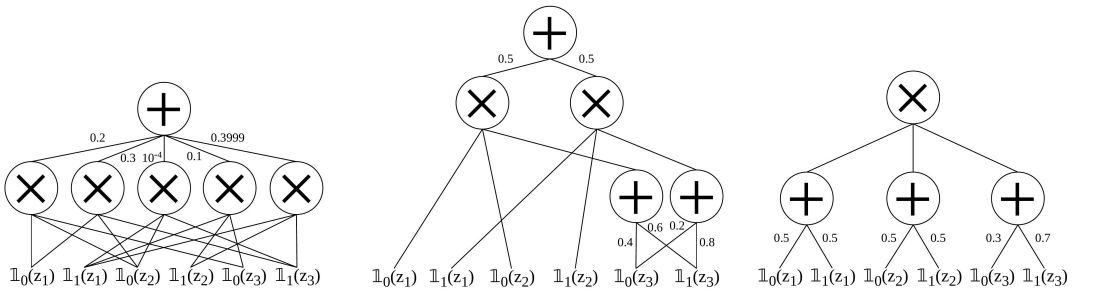
can compress data by using an example distribution, shown in Table 5.1. In this example, we focus on compressing this table's data and also to slightly generalise by removing irrelevant noise from the data.

*Table 5.1: Example joint probability, where $z_1$, $z_2$ and $z_3$ are discrete variables, having 2 states each. Note the slight noise added to the data, as seen in the* 0.0001 *and* 0.3999 *values.*

| $z_1$ | $z_2$ | $z_3$ | $p(z_1, z_2, z_3)$ |
|---|---|---|---|
| 0 | 0 | 0 | 0.2 |
| 0 | 0 | 1 | 0.3 |
| 1 | 0 | 1 | 0.0001 |
| 1 | 1 | 0 | 0.1 |
| 1 | 1 | 1 | 0.3999 |

Table 5.1 represents a distribution over 3 random variables. We would like our model to represent this distribution in a compact form. The probability entry with value $10^{-4}$ can be assumed to be due to noise and we would therefore not like to model this value. Constructing models that generalise well is especially hard when noise is introduced. If one does not compress the data enough, the model overfits to the data and does not generalise well. Overfitting occurs when the network starts to model the noise in a dataset, which decreases the network's accuracy on new unseen test data generated from the same underlying distribution. If the model compresses the data too much, the network starts to underfit (modelling too little of the underlying distribution) and also does not generalise well. These three stages of compression of the data in Table 5.1 are illustrated in Figure 5.1.

The goal of a learning algorithm is to construct an SPN that generalises well to unseen data. To do so the model must not underfit nor overfit to training data. The problem of underfitting can be solved relatively easily, as it is the direct objective of the optimisation algorithm, as discussed in the next section. By increasing the model's complexity (number of nodes and connections) and allowing the model to train longer, the model should be able to better represent the training data.




*(a) Not compressed (overfit)* *(b) Good compression* *(c) Overcompressed (underfit)*

*Figure 5.1: The three images represent different networks that try to represent the probability distribution described in Table 5.1. In image (a), the SPN represents the full probability table with its noise and therefore overfits to the data. In image (b) the network compresses the data by just the right amount to remove the noise, but still models the underlying distribution. Image (c) illustrates a fully compressed network, with independent variables. This network is therefore too compressed and does not model the underlying distribution well, and will not generalise to new unseen data. Note that image (b) is also factored into a compact form, compared to image (a), which reduces the number of calculations needed in obtaining a probability value.*

A generally more challenging problem in machine learning is to avoid overfitting to data. This is because a basic optimiser usually just wants to decrease an error between the model's predictions and the data. If the optimiser trains a large model for too long it can start to represent the noise in the data, leading to overfitting. We can combat overfitting by including a prior probability distribution over possible models, which is further elaborated on in the next section. We generally rely on two methods to train SPNs, called parameter learning and structure learning. In parameter learning we assume we already have a compact network structure e.g. Figure 5.1 (b), but we still need to learn the parameters that correctly represent the data and which generalises well. In the case of iterative structure learning, as is performed by the SET-SPN and MIXEDCLONES algorithms, we start with a completely underfit network e.g. Figure 5.1 (c). We then iteratively expand the network until we reach a satisfactory network structure and parameters as again seen in e.g. Figure 5.1 (b). We next describe the mathematics of the optimisation processes used the train a model for both parameter and structure learning.

### 5.1.2 Mathematical description

As described in Section 3.2, the goal of the learning algorithms is to discover the best model to represent $N$ data vectors, individually denoted by $\mathbf{x}_n$ and $\mathbf{y}_n$, while also generalising well to unseen data. Here $\mathbf{x}_n$ and $\mathbf{y}_n$ individually represent values for the input ($\mathbf{x}$) and output ($\mathbf{y}$) random variables at data point $n$. The entire dataset that the network is trained from is represented by $\mathbf{x}_{1:N}$ and $\mathbf{y}_{1:N}$, where $N$ is the number of data points. We derived from Section 3.2 that the parameters we want that achieve the maximum likelihood to be equal to

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}}\, p_m(\boldsymbol{\theta}_m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)}, \tag{5.1}$$

where $m$ represents the model (structure) being used and $\boldsymbol{\theta}_m$ the parameters of that model structure. It is again important to note that equation (5.1) holds in the generative and discriminative setting, as $Z$ serves as a normalising term in the case of discriminative training ($g = False$). We also derived the optimal model we want that achieves the maximum likelihood for model selection is

$$\hat{m} = \underset{m}{\operatorname{argmax}}\, p(m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)}. \tag{5.2}$$

The terms $p(m)$ and $p_m(\boldsymbol{\theta}_m)$ represent the prior probability distributions over possible model structures and over the parameters of that models. These prior terms can help regulate the network against overfitting and decrease training times. In this work, we use the probability term $p(m)$ to prevent the network from becoming too large by making the probability of larger networks less likely than smaller networks, as is done in Section 6.1.2. The probability term $p_m(\boldsymbol{\theta}_m)$ is used to regulate weight values to prevent them from becoming to large and also help decreasing training times. Two mathematical representations of $p_m(\boldsymbol{\theta}_m)$ and $p(m)$ are also provided in Section 5.2.1 and 6.1.2.

We would now like to convert equations (5.1) and (5.2) to their corresponding logarithmic forms. This is performed to avoid numerical instabilities. All SPN calculations are also performed in logarithmic form as is described in Section 4.2. We know that

$$\underset{\mathbf{z}}{\operatorname{argmax}}\, f(\mathbf{z}) = \underset{\mathbf{z}}{\operatorname{argmax}} \ln(f(\mathbf{z})), \tag{5.3}$$

where $f$ is a valid function, with the condition $f(\mathbf{z}) > 0$, and $\mathbf{z}$ is a set of variables. Equation (5.3) holds due to the logarithmic function being a monotonically increasing function. The output of the logarithmic of a function always increases as the function itself increases. Therefore, the maximum value of the function is at the same point, in terms of $\mathbf{z}$, as the maximum value of logarithm of the function is. We can, therefore, rewrite equation (5.1) to form

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}} \ln \left( p_m(\boldsymbol{\theta}_m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n | \boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n | \boldsymbol{\theta}_m), g)} \right). \tag{5.4}$$

Equation (5.4) can now be further simplified to form the final parameter optimisation objective of

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}} \ln(p_m(\boldsymbol{\theta}_m)) + \sum_n \ln(p_m(\mathbf{y}_n, \mathbf{x}_n | \boldsymbol{\theta}_m)) - \ln(Z(p_m(\mathbf{x}_n | \boldsymbol{\theta}_m), g)). \tag{5.5}$$

The logarithmic equivalent of our model selection goal can similarly be rewritten to be

$$\hat{m} = \underset{m}{\operatorname{argmax}} \ln(p(m)) + \sum_n \ln(p_m(\mathbf{y}_n, \mathbf{x}_n | \boldsymbol{\theta}_m)) - \ln(Z(p_m(\mathbf{x}_n | \boldsymbol{\theta}_m), g)). \tag{5.6}$$

In the generative setting the term $Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)$ is equal to one and $\ln(Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g))$ becomes zero. We will now look at ways of finding a suitable SPN $(m, \boldsymbol{\theta}_m)$ using parameter learning and later using structure learning as well. In the next section, we look at the mathematics behind parameter learning methods to train an SPN from data.

## 5.2 Parameter learning algorithms

Parameter learning in SPNs involves first defining a valid starting network structure and then learning the weights of the network that best fit some data distribution. The goal of a parameter learning algorithm is to maximise the likelihood expression presented in equation (5.5). The main parameter learning algorithm used in this work is the gradient descent algorithm. Gradient descent parameter learning has achieved state of the art results on many SPN benchmark datasets (Van de Wolfshaar & Pronobis, 2019) in the generative and discriminative setting. The first-order gradient descent algorithm is used in many different optimisation problems and is the standard method for training deep neural networks. There are also second-order gradient descent methods, which we briefly

discuss in Section 5.2.1.5. Another popular generative parameter learning algorithm that is used in SPNs is the Expectation Maximisation algorithm, as proposed in Poon & Domingos (2011).

These parameter learning algorithms are also used in structure learning as a way to fine-tune weights, and is tested and discussed in more detail in Section 7.2.2.1. We now start by first investigating how first-order gradient descent can be used to train SPN parameters.

### 5.2.1 Gradient descent optimisation

As the name suggests, gradient descent relies on gradients to indicate in which direction to change a weight to decrease the loss of a network. The loss function that we would like to minimise in parameter learning is defined as

$$\text{log loss} = J = -[\ln(p_m(\boldsymbol{\theta}_m)) + \sum_n \ln(p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)) - \ln(Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g))]. \tag{5.7}$$

It is important to note that the log loss is simply the negative of the log likelihood function we maximise over in equation (5.5). In this case we could also have maximised the log likelihood directly (gradient ascent). We, however, opt to describe the gradient descent algorithm as it is closely related to the deep learning literature. The prior log probability of our weights in our network is specified by $\ln(p_m(\boldsymbol{\theta}_m))$. We would like this term to help ensure the parameters of the network stay valid. Firstly, we can easily set all weight values to not go lower than 0. We would also like to have the sum of each sum node's weights to be approximately equal to one. SPNs still output valid, but scaled, probability values when these weights are not normalised, but the SPN's calculations might encounter numerical instabilities if the sum of the weights become too large or small. We would therefore like the prior log probability distribution term to regulate the weights in the network so that they remain approximately normalised. One solution is to define a regulating term as follows

$$\text{reg} = -\frac{\lambda}{N_s} \sum_i \sum_j (1 - \sum_k w_{ijk})^2, \tag{5.8}$$

where $\lambda$ tells the optimiser how much it should focus on making sure the sum of each sum node's weights is close to 1. The $i$ index represents the layer we are currently working

with in the network. The $j$ index represents a node in layer $i$. The value $k$ represents the index of a weight in node $ij$. The reason we use this notation will become more apparent once we describe how to calculate the gradients in Section 5.2.1.1. It is important to note that for this regularisation term we are only summing over nodes $ij$ that are sum nodes, even though there are other node types which also have parameters, e.g. Gaussian nodes. The value $N_s$ represents the number of sum nodes in the network. The reason we divide by $N_s$ is, because it is good practice to define $\lambda$ in such a way that the same value can be used for differently sized networks. Therefore, by divide by $N_s$ we are effectively using the average sum node weight error as the regularisation term, which is independent from the number of nodes in the network. However, this regularisation term is not a valid log probability term yet. We can now define

$$p_m(\boldsymbol{\theta}_m) = \frac{e^{\text{reg}}}{\int_{-\infty}^{\infty} e^{\text{reg}} d\boldsymbol{\theta}_m}, \tag{5.9}$$

which converted to its logarithmic form becomes

$$\ln(p_m(\boldsymbol{\theta}_m)) = \text{reg} - \ln\left(\int_{-\infty}^{\infty} e^{\text{reg}} d\boldsymbol{\theta}_m\right). \tag{5.10}$$

The normalisation term of $\ln\left(\int_{-\infty}^{\infty} e^{\text{reg}} d\boldsymbol{\theta}_m)\right)$ is, however, constant for different values of $\boldsymbol{\theta}_m$ and therefore does not have to be included in the optimisation process. We can now also replace $\boldsymbol{\theta}_m$ with $\mathbf{w}$, where $\mathbf{w}$ is a vector containing all the weights in the network. Equation (5.7) can now be rewritten as

$$J = \frac{\lambda}{N_s} \sum_i \sum_j (1 - \sum_k w_{ijk})^2 + \frac{1}{N} \sum_n \ln(Z(p_m(\mathbf{x}_n|\mathbf{w}), g)) - \ln(p(\mathbf{y}_n, \mathbf{x}_n|\mathbf{w})). \tag{5.11}$$

Note that we also want to work with the average data point loss, as this means the learning rate ($\alpha$), defined below, does not need to be scaled according to the size of each dataset. We, therefore, include the $\frac{1}{N}$ term to make this possible. This $\frac{1}{N}$ term, however, does not change where the best model parameters are and therefore will result in the same optimal parameters as minimising equation (5.7). The term is only included to make weight updating simpler. A lower value for equation (5.11) means we have an approximately better model to fit the data. Figure 5.2 illustrates that by incrementally moving down a slope, defined by $\frac{\partial J}{\partial w_{ijk}}$, the global loss of the network can be decreased.

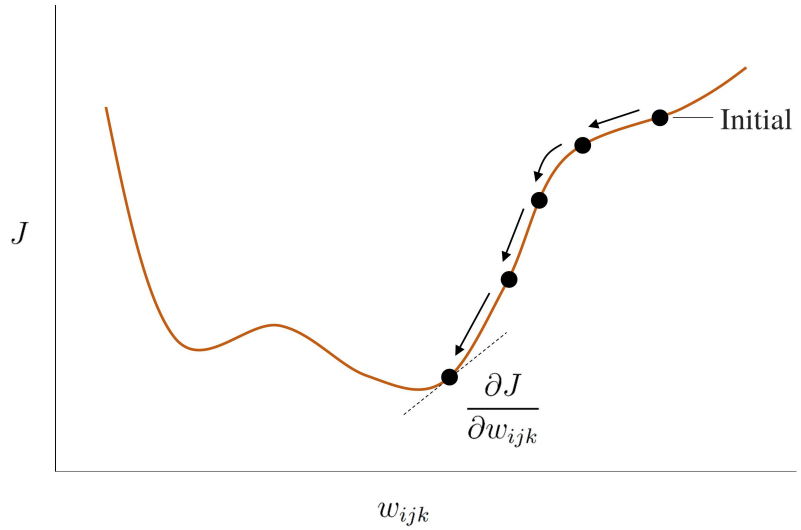



*Figure 5.2: Visualisation of the gradient descent process. Each arrow indicates an incremental update to the weight, as instructed by the optimiser, to try and minimise the loss function.*

The basic gradient descent optimiser is defined as

$$w^*_{ijk} = w_{ijk} - \alpha \frac{\partial J}{\partial w_{ijk}}, \tag{5.12}$$

where $w^*_{ijk}$ and $w_{ijk}$ are the new and old weight values at indices $ijk$. The value $\alpha$ represents the learning rate of the optimiser. This value controls how large the step size of each weight update is. If this value is too large, the network might not converge to an answer, and if it is too small the network might take too long to converge to a solution. It is thus important to test different learning rates and find which one works well for a given training setting. A good $\alpha$ value is usually between 0.01 and 1.0, although there is no fixed standard. There are also more advanced optimisers that dynamically assign different learning rates to each weight, which is further discussed in Section 5.2.1.5. The value $J$ is the log loss of the network, defined in equation (5.11). Equation (5.12) therefore takes the gradient of the loss, with respect to $w_{ijk}$, and updates the weight in the negative gradient direction. Algorithm 2 provides an overview of the gradient descent process applied to SPNs.

**Algorithm 2** Gradient descent parameter learning.

**Data:** Training data.

**Result:** Network with trained weights.

Randomly initialise network weights.

```
for predefined number of epochs do
    Inference on training data.
    Backpropagate loss gradients.
    for i in number of layers do
        for j in number of nodes in layer do
            for k in number of weights of node do
                w*_ijk = w_ijk − α ∂J/∂w_ijk
            end
        end
    end
end
```

We now need to calculate all the $\frac{\partial J}{\partial w_{ijk}}$ values, representing the gradients with respect to each weight, as described in the next section.

#### 5.2.1.1 Calculating gradients

Using equation (5.11), we can calculate the $\frac{\partial J}{\partial w_{ijk}}$ values to be

$$\frac{\partial J}{\partial w_{ijk}} = \frac{\lambda}{N_s}\frac{\partial(1-\sum_q w_{ijq})^2}{\partial w_{ijk}} + \frac{1}{N}\sum_n \frac{\partial \ln(Z(p_m(\mathbf{x}_n|\mathbf{w}),g))}{\partial w_{ijk}} - \frac{\partial \ln(p(\mathbf{y}_n,\mathbf{x}_n|\mathbf{w}))}{\partial w_{ijk}}, \quad (5.13)$$

where $w_{ijk}$ is in layer $i$ and one of the weights of node $ij$. The gradient of the regularisation term can be calculated relatively easily to be

$$\frac{\partial(1-\sum_q w_{ijq})^2}{\partial w_{ijk}} = 2w_{ijk}\left[\left(\sum_q w_{ijq}\right) - 1\right]. \quad (5.14)$$

Similarly the gradient value of the normalising term can be calculated as

$$\frac{\partial \ln(Z(p_m(\mathbf{x}_n|\mathbf{w}),g))}{\partial w_{ijk}} = \begin{cases} 0 & \text{if } g = \text{True} \\ \frac{\partial \ln(p(\mathbf{x}_n|\mathbf{w}))}{\partial w_{ijk}} & \text{if } g = \text{False} \end{cases}. \quad (5.15)$$

The challenge lies in calculating the gradient values $\frac{\partial \ln(p(\mathbf{x}_n|\mathbf{w}))}{\partial w_{ijk}}$ and $\frac{\partial \ln(p(\mathbf{y}_n,\mathbf{x}_n|\mathbf{w}))}{\partial w_{ijk}}$. Both these gradients can be calculated using the same method as both of them are just the gradient of the logarithmic output of the network with respect to some weight. The only difference in the computations of these two gradient values is the data passed through the network. For the gradient value of $\frac{\partial \ln(p(\mathbf{y}_n,\mathbf{x}_n|\mathbf{w}))}{\partial w_{ijk}}$, the joint probability of the data is evaluated by the network and then the gradient gets calculated. For the gradient value of $\frac{\partial \ln(p(\mathbf{x}_n|\mathbf{w}))}{\partial w_{ijk}}$, the marginal probability of the input data is passed through the network and then the gradient value gets calculated. Therefore, we are interested in calculating the general gradient value of $\frac{\partial \ln(p_r)}{\partial w_{ijk}}$, where $\ln(p_r)$ is the logarithmic output of the network. To calculate this gradient value in an efficient manner we rely on the chain rule. The chain rule is a technique that can be used to find the derivative of composite functions in an efficient manner. SPNs consist of different layers with sets of nodes in each layer. We use the symbol $N_i$ to indicate the number of layers in a network. By calculating each layer one at a time, from layer 1 to layer $N_i$, the probability output of a network, for a given query, can be determined. The output of the network is presented at the output of layer $N_i$. We can now also calculate the $\frac{\partial \ln(p_r)}{\partial w_{ijk}}$ values by sequentially calculating gradients from layer $N_i$ back to layer 1. To calculate these gradient values we rely on the chain rule. The chain rule applies if we know that

$$p_r = (\mathbf{p}_{N_i} \circ \mathbf{p}_{N_i-1} \circ \cdots \circ \mathbf{p}_i)(w_{ijk}) = \mathbf{p}_{N_i}(\mathbf{p}_{N_i-1}(\ldots(\mathbf{p}_i(w_{ijk}))\ldots)), \tag{5.16}$$

where layer $i$ is the layer which has the node with weight value $w_{ijk}$ in it. Here, $o$ represents function composition. There are some SPN setups where one layer relies on more than one previous layer to calculate its output. The backpropogation algorithm can also be used in this case, but for simplicity we assume layers only depend on the previous layer, as with most randomly generated SPNs. The vector $\mathbf{p}_i$, represents the probability outputs of all the nodes in layer $i$ combined into one vector. If equation (5.16) is true, as is the case in SPNs, it is also true in the logarithmic domain, where every probability is just converted to its logarithmic form. We also know that the value $\ln(p_r)$ is the output of the first, and only, entry in the vector $\ln(\mathbf{p}_{N_i})$. The chain rule states that we can, therefore, calculate the gradient values, in logarithmic form using numerator-layout notation, as follows

$$\frac{\partial \ln(p_r)}{\partial w_{ijk}} = \frac{\partial \ln(\mathbf{p}_{N_i})}{\partial \ln(\mathbf{p}_{N_i-1})} \frac{\partial \ln(\mathbf{p}_{N_i-1})}{\partial \ln(\mathbf{p}_{N_i-2})} \cdots \frac{\partial \ln(\mathbf{p}_i)}{\partial w_{ijk}}. \tag{5.17}$$

Equation 5.17 now provides a method for calculating the logarithm of the gradient of the output of the network with respect to any weight. It is important to notice that the gradients of each layer can be calculated in an efficient manner using

$$\frac{\partial \ln(p_r)}{\partial w_{ijk}} = \frac{\partial \ln(\mathbf{p}_{N_i})}{\partial \ln(\mathbf{p}_{i+1})} \frac{\partial \ln(\mathbf{p}_{i+1})}{\partial \ln(\mathbf{p}_i)} \frac{\partial \ln(\mathbf{p}_i)}{\partial w_{ijk}}, \tag{5.18}$$

where $\frac{\partial \ln(\mathbf{p}_{N_i})}{\partial \ln(\mathbf{p}_{i+1})}$ represents the gradient vector calculated in the previous layer $(i+1)$. Therefore, by re-using the gradient calculations of the previously calculated layer $(i+1)$, the next layer's gradient values can be calculated. If we now want to calculate all the gradient values in the network we simply calculate the values $\frac{\partial \ln(\mathbf{p}_{i+1})}{\partial \ln(\mathbf{p}_i)}$ and $\frac{\partial \ln(\mathbf{p}_i)}{\partial w_{ijk}}$ from layer $N_i - 1$ back to layer 1. This method of starting at the end of the network and calculating each layer's gradients backwards through the network is called the backpropagation algorithm. In this way we can re-use computations performed in the previous layer, which allows the gradient values to be calculated in time linear to the size of the network. For the output layer $(N_i)$, both the vectors $\frac{\partial \ln(\mathbf{p}_{N_i})}{\partial \ln(\mathbf{p}_{i+1})}$ and $\frac{\partial \ln(\mathbf{p}_{i+1})}{\partial \ln(\mathbf{p}_i)}$, as seen in equation (5.18), are vectors where every entry has a value of 1, as there is no layer above it (e.g. no layer at index $N_i + 1$). If the output layer is a sum node only the gradient value $\frac{\partial \ln(\mathbf{p}_{N_i})}{\partial w_{N_i 1k}}$ needs to be calculated for every weight in that node.

Because the vector $\frac{\partial \ln(\mathbf{p}_{i+1})}{\partial \ln(\mathbf{p}_i)}$ is simply a concatenation of gradient values for layer $i$, we can calculate the gradient values $\frac{\partial \ln(\mathbf{p}_{i+1})}{\partial \ln(p_{ij})}$ individually, and then concatenated them together at the end. Each value in the vector $\frac{\partial \ln(\mathbf{p}_{i+1})}{\partial \ln(p_{ij})}$ is also equal to $\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})}$, where $j_p$ and $j_c$ represents the $j$ indices of nodes in layer $i+1$ (parent) and layer $i$ (child), respectively. Similarly, every value in $\frac{\partial \ln(\mathbf{p}_i)}{\partial w_{ijk}}$ can be represented by $\frac{\ln(p_{ij_p})}{\partial w_{ijk}}$, where $j_p$ is the index of the vector $\mathbf{p}_i$. We can now deduce that

$$\frac{\partial \ln(p_{ij_p})}{\partial w_{ijk}} = \begin{cases} \frac{\partial \ln(p_{ij})}{\partial w_{ijk}} & \text{if } j_p = j \\ 0 & \text{Otherwise} \end{cases}, \tag{5.19}$$

as the value $w_{ijk}$ only directly influences the output of the node at index $ij$. We still need to describe how to calculate $\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})}$ for parent sum and product nodes, in layer $i+1$, as well as $\frac{\partial \ln(p_{ij})}{\partial w_{ijk}}$ for sum nodes. We also need to calculate $\frac{\partial \ln(p_{ij})}{\partial w_{ijk}}$ for the Gaussian leaf distribution, as it is a common leaf distribution to use. Leaf distributions cannot be parent nodes and therefore the value $\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})}$ does not need to be calculated for them.

#### 5.2.1.2 Product node gradients

In the logarithmic domain, product nodes are represented by additions over the logarithmic values of its children. Before we define the output of a product node, in terms of network layers, we first define a vector $\mathbf{c}(i+1, j_p)$ that represents all the $j$ indices of children of node $(i+1)j_p$. The first entry of this vector is, therefore, represented by $c(i+1, j_p)_1$. We can, therefore, use this function to find the location of children nodes in the layer underneath the parent node. Adapted from equation (4.2), the output of a parent product node $((i+1)j_p)$ can be calculated using

$$\ln(p_{(i+1)j_p}) = \ln(p_{ic(i+1,j_p)_1}) + \cdots + \ln(p_{ic(i+1,j_p)_K}), \tag{5.20}$$

where $K$ is the number of children of the product node and $\ln(p_{ic(i+1,j_p)_K})$ represents the logarithmic output of a child of the product node at index $K$. The logarithm of the output of the product node is represented by $\ln(p_{(i+1)j_p})$. The gradient of a product node $((i+1)j_p)$ with respect to each node in the previous layer $(ij_c)$ is described by

$$\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{(ij_c)})} = \begin{cases} 1 & j_c \text{ in } \mathbf{c}(i+1, j_p) \\ 0 & \text{Otherwise} \end{cases}. \tag{5.21}$$

Therefore it is fairly easy to backpropagate the gradient of the network's loss through a product node.

#### 5.2.1.3 Sum node gradients

In the logarithmic domain, sum nodes use the log sum exponent trick to calculate their outputs to ensure numerical stability. As adapted from equation (4.3), the output of a sum node $((i+1)j_p)$ can be calculated using

$$\ln(p_{(i+1)j_p}) = \ln[e^{\ln(w_{(i+1)j_p1})+\ln(p_{ic(i+1,j_p)_1})-M} + \cdots + e^{\ln(w_{(i+1)j_pK})+\ln(p_{ic(i+1,j_p)_K})-M}] + M, \tag{5.22}$$

where $K$ is the number of children of the sum node. The value $M$, as adapted from equation (4.4), is

$$M = \max_{k \in [1,K]} \ln(w_{(i+1)j_pk}) + \ln(p_{ic(i+1,j_p)_k}). \tag{5.23}$$

We now define $c_p(i+1, j_p, j_c)$ as the index value in vector $\mathbf{c}(i+1, j_p)$ where the entry value $j_c$ is located. This function, therefore, gives the index of the weight connecting the

parent node with its child node. The gradient of a sum node, at index $(i+1)j_p$, with respect to a node in the previous layer, at index $ij_c$, is described by

$$\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})} = \begin{cases} e^{\ln(w_{(i+1)j_p c_p(i+1,j_p,j_c)})+\ln(p_{ij_c})-\ln(p_{(i+1)j_p})} & j_c \text{ in } \mathbf{c}(i+1,j_p) \\ 0 & \text{Otherwise} \end{cases}. \tag{5.24}$$

Equation (5.24) is written in this form as it is also numerical stable to do this calculation using 64-bit floating point numbers. Deriving the gradient with respect to the weight of a sum node $(ij)$ at index $k$ leads to

$$\frac{\partial \ln(p_{ij})}{\partial w_{ijk}} = \frac{1}{w_{ijk}} \frac{\partial \ln(p_{ij})}{\partial \ln(p_{(i-1)c(i,j)_k})}. \tag{5.25}$$

A more detailed derivation of equation (5.24) and equation (5.25) is provided in Appendix A.

#### 5.2.1.4 Gaussian node gradients

A commonly used leaf distribution in SPNs is the univariate Gaussian distribution, which is defined over a continuous random variable. A Gaussian node has two parameters namely, the mean, represented by $\mu$, and standard deviation, which is represented by $\sigma$. We would also like to update these values in the gradient descent process. The log domain output of the univariate Gaussian function is represented by

$$ln(f(z|\mu,\sigma)) = -\frac{1}{2}ln(2\pi) - ln(\sigma) - \frac{(z-\mu)^2}{2\sigma^2}, \tag{5.26}$$

where $z$ represents the state value of the continuous random variable that the Gaussian is defined over. For the Gaussian node, at index $ij$, we need to calculate $\frac{\partial \ln(p_{ij})}{\partial w_{ijk}}$, which simplifies to $\frac{\partial \ln(f(z|\mu,\sigma))}{\partial \mu}$ and $\frac{\partial \ln(f(z|\mu,\sigma))}{\partial \sigma}$. By deriving equation (5.26) with respect to $\mu$, we get

$$\frac{\partial \ln(f(z|\mu,\sigma))}{\partial \mu} = \frac{z-\mu}{\sigma^2}. \tag{5.27}$$

Similarly, we also derive equation (5.26) with respect to $\sigma$, which yields

$$\frac{\partial \ln(f(z|\mu,\sigma))}{\partial \sigma} = \frac{(z-\mu)^2}{\sigma^3} - \frac{1}{\sigma}. \tag{5.28}$$

We have now derived all the gradients needed to apply the chain rule to an SPN. The backpropagation algorithm is an efficient way of calculation these gradients, by

using Dynamic Programming (DP). Dynamic programming stores some computations, that are used more than once, in memory after the first computation is conducted. These computations can then be retrieved without the need to recompute them, therefore saving processing time. We, therefore, can now calculate the gradient value $\frac{\partial J}{\partial w_{ijk}}$ for every weight in the network. This loss derivative can now, in turn, be used to update the weights in the network using a basic optimiser that follows the update rule defined in equation (5.12). There are also more advanced optimisers that can be used, given one has the loss derivative values with respect to each weight. These optimisers are discussed in the next section.

#### 5.2.1.5 The optimiser

It is worth mentioning that there are also second-order gradient descent methods, for example, the Newton Raphson algorithm. When these second-order methods can be implemented exactly in a network, they generally lead to faster convergence than first-order gradient descent methods. Second-order gradient methods are thus regularly used for training small networks like those used in logistic regression. For larger networks, these second-order methods usually cannot be implemented in their exact form and approximations need to be used. These approximations, however, lead to degraded results compared to the exact methods. Therefore, due to the size of SPNs and DNNs, state of the art results for both models rely on first-order gradient descent methods. One optimiser that is extensively used in training these networks is the Adam optimiser, which indirectly incorporates some additional second-order gradient information as well (Kingma & Ba, 2015). The Adam optimiser also assigns a different learning rate to each weight, which speeds up training times. Another trick that is employed to increase learning speeds is to approximate $\frac{\partial J}{\partial w_{ijk}}$ with smaller batches of data. Therefore, not all the data points are used in each weight update. This allows the optimiser to do more weight updates and therefore converge faster to better network configurations.

### 5.2.2 Expectation maximisation

Another popular parameter learning algorithm, derived from the probabilistic modelling literature, is the Expectation Maximisation (EM) algorithm. This algorithm has been extensively used to train generative models like Gaussian Mixture Models (GMMs) and

Hidden Markov Models (HMMs). The EM algorithm is a method used to try to find MLE (or MAP) for models.

Let us start by re-writing the output of an SPN into the form

$$p_m(\mathbf{y}, \mathbf{x}|\boldsymbol{\theta}_m) = \sum_{\forall \mathbf{h}} p_m(\mathbf{y}, \mathbf{x}, \mathbf{h}|\boldsymbol{\theta}_m), \tag{5.29}$$

where $\mathbf{y}$ and $\mathbf{x}$, respectively, represent the output and input random variables. Here $\mathbf{h}$ represents the set of hidden variables in the network, which are being marginalised out. The vector $\boldsymbol{\theta}_m$ again represents all the parameters for model $m$. Every sum node in the network can be seen as performing marginalisation over the hidden variable represented by that node. The number of states a hidden variable has is, therefore, equal to the number of children the corresponding sum node has.

As derived in equation (5.5) the maximise operation we want to perform for parameter learning is

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}} \ln(p_m(\boldsymbol{\theta}_m)) + \sum_n \ln(p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)) - \ln(Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)), \tag{5.30}$$

where $\hat{\boldsymbol{\theta}}_m$ is the optimal set of parameters we would like to find. Here $\ln(Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g))$ is the logarithm of a normalising term in the discriminative setting. This EM algorithm works in the generative setting and therefore $\ln(Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g))$ is equal to zero. In EM the weights of the network is also normalised after each update. As no regularisation is needed, we assume a uniform prior distribution $(p_m(\boldsymbol{\theta}_m))$, which does not change for different $\boldsymbol{\theta}_m$ and can be discarded. By substituting in equation (5.29), the new simplified optimisation goal of the EM algorithm can now be written as

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}} \sum_n \ln \left( \sum_{\forall \mathbf{h}} p(\mathbf{y}_n, \mathbf{x}_n, \mathbf{h}|\boldsymbol{\theta}_m) \right). \tag{5.31}$$

The EM algorithm now iteratively updates the parameters $(\boldsymbol{\theta}_m)$ of the network starting with some initial parameters. In EM an update step consists of finding

$$\boldsymbol{\theta}_m^* = \underset{\boldsymbol{\theta}_m^-}{\operatorname{argmax}} \sum_n \sum_{\forall \mathbf{h}} p(\mathbf{h}|\mathbf{y}_n, \mathbf{x}_n, \boldsymbol{\theta}_m) \ln \left( p(\mathbf{y}_n, \mathbf{x}_n, \mathbf{h}|\boldsymbol{\theta}_m^-) \right), \tag{5.32}$$

where $\boldsymbol{\theta}_m^*$ and $\boldsymbol{\theta}_m$ represent the new and current parameters of the network, respectively. The vector $\boldsymbol{\theta}_m^-$ represents the parameters we are searching over.

#### 5.2.2.1 Sum node hidden variables

In the case of a sum node, the parameters of this node is in the form of weight values. Maximising equation (5.32) for weights of a discrete hidden variable, e.g. sum node weights, was derived in Desana & Schnörr (2016) to be

$$w_{ij}^{*} = \frac{w_{ij}[\sum_n \frac{1}{p_r(y_n,x_n)} \frac{\partial p_r(y_n,x_n)}{\partial p_i(y_n,x_n)} p_j(y_n,x_n)]}{\sum_{q\in\text{children}(i)} w_{iq}[\sum_n \frac{1}{p_r(y_n,x_n)} \frac{\partial p_r(y_n,x_n)}{\partial p_i(y_n,x_n)} p_q(y_n,x_n)]}, \tag{5.33}$$

where $w_{ij}$ represents the weight connecting sum node at index $i$ with its child node at index $j$. The value $w_{ij}^{*}$, represents the updated value of weight $w_{ij}$. In equation (5.33) $p_r(y_n, x_n)$ represents the probability output of the root node for data point $n$. The value $p_i(y_n, x_n)$ represents the output of the sum node at index $i$. The values $p_j(y_n, x_n)$ and $p_q(y_n, x_n)$ represent two of the sum node's children node probability outputs at index $j$ and $q$, respectively, for the given data point $n$. Due to the probability output of a network possibly being extremely small, numerical underflow might occur if we directly calculate equation (5.33) using 64-bit numbers. Therefore, all the multiplications should first be performed in the logarithmic domain, before computing the final weight updates. This can be done be rewriting equation (5.33) as follows

$$w_{ij}^{*} = \frac{w_{ij}\left[\sum_n e^{\ln\left(\frac{\partial p_r(y_n,x_n)}{\partial p_i(y_n,x_n)}\right) - \ln(p_r(y_n,x_n)) + \ln(p_j(y_n,x_n))}\right]}{\sum_{q\in\text{children}(i)} w_{iq}\left[\sum_n e^{\ln\left(\frac{\partial p_r(y_n,x_n)}{\partial p_i(y_n,x_n)}\right) - \ln(p_r(y_n,x_n)) + \ln(p_q(y_n,x_n))}\right]}. \tag{5.34}$$

#### 5.2.2.2 Gaussian node hidden variables

For SPNs we are especially interested in using univariate Gaussian leaf distributions. The update rule for univariate Gaussian distributions was also derived in Desana & Schnörr (2016). We start by calculating responsibility values

$$r_{ln} = \frac{p_l(y_n, x_n)}{p_r(y_n, x_n)} \frac{\partial p_r(y_n, x_n)}{\partial p_l(y_n, x_n)}, \tag{5.35}$$

where $r_{ln}$ indicates how responsible the leaf node at index $l$ is for representing data point $n$ for the current update step. Here $p_l$ represents the probability output for that leaf node. The parameters of a univariate Gaussian distribution is in the form of a mean ($\mu$)

and a standard deviation ($\sigma$) value. We can estimate the mean of the univariate Gaussian to be

$$\mu_l^* = \frac{\sum_n r_{ln} z_n}{\sum_n r_{ln}}, \tag{5.36}$$

where $z_n$ is the state value, for the random variable the Gaussian is defined over, for data point $n$. The value $\mu_l^*$ represents the updated mean value of the leaf Gaussian distribution at index $l$. The new standard deviation of the Gaussian node can be estimated using

$$\sigma_l^* = \sqrt{\frac{\sum_n r_{ln}(z_n - \mu_l^*)^2}{\sum_n r_{ln}}}, \tag{5.37}$$

where $\sigma_l^*$ represents the updated standard deviation value of the leaf Gaussian distribution at index $l$. We now have suitable update rules to update the parameters of a network using EM. An advantage of using EM is that it has no learning rate that has to be specified by a user. These EM updates can now be applied iteratively until the parameters of a network converge or some maximum iteration step is reached.

## 5.3 Conclusion

In this chapter, we investigated training SPNs directly from data. We investigated two parameter learning algorithms, namely gradient descent and expectation maximisation. We found that all the gradient values in the gradient descent process can be calculated in time linear to the size of the network. We, therefore, found that gradient descent could easily be used to train SPNs in the generative and discriminative setting and with discrete and continuous data types. An advantage of the EM algorithm is that it requires no learning rate, which might allow for faster training times in the generative setting that gradient descent can deliver.

In this chapter, we focussed on learning the parameters of an SPN and investigate learning the entire structure of an SPN directly from data.

# Chapter 6

# Structure learning: SET-SPN

The goal of structure learning is to discover a model structure that maximises the optimisation goal for structure learning (model selection). The optimisation goal for structure learning, as defined in equation (3.18), is

$$\hat{m} = \underset{m}{\operatorname{argmax}}\, p(m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)}, \tag{6.1}$$

where $\hat{m}$ is the best model to fit our data. We use this optimisation goal, expressed in the linear probability domain, as it is simpler to explain structure learning in this domain. In equation (6.1), we assume we have a set of parameters for each model. We can, however, iterate between selecting better models using equation (6.1) and updating the parameters of the best models using equation (3.17), which is rewritten as

$$\hat{\boldsymbol{\theta}}_m = \underset{\boldsymbol{\theta}_m}{\operatorname{argmax}} \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)}, \tag{6.2}$$

where $\hat{\boldsymbol{\theta}}_m$ is the optimal parameters for that model. We removed the $p_m(\boldsymbol{\theta}_m)$ term from equation (3.17) as we assume a uniform distribution over all the weights. One of the reasons we might have wanted to use parameter regularisation is to ensure that each sum node's weights sum to one. However, weights are always normalised in our structure learning approach and therefore we do not need regularisation to ensure normalisation. We also do not consider some normalised weight combinations to be more likely than others and therefore a uniform distribution is selected. As we usually cannot solve equation (6.2) directly, we try to iteratively optimise both equations (6.1) and (6.2) in our

structure learning approach. This is generally a harder problem than just learning the parameters of a network. We, therefore, create our final optimisation goal by combining equations (6.1) and (6.2) to form

$$\left(\hat{m}, \hat{\boldsymbol{\theta}}_{\hat{m}}\right) = \underset{m,\boldsymbol{\theta}_m}{\operatorname{argmax}}\, p(m) \prod_n \frac{p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)}, \tag{6.3}$$

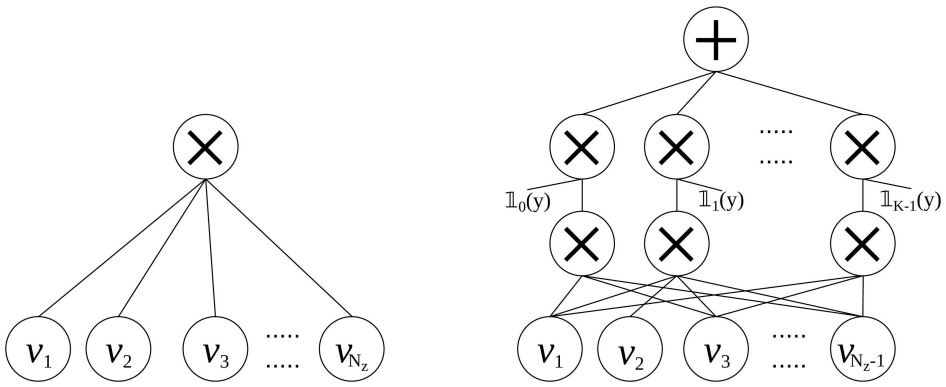
where $\hat{\boldsymbol{\theta}}_{\hat{m}}$ is the optimal parameters for the optimal model $\hat{m}$. Notice that if we optimise equation (6.3) just with respect to $m$ or $\boldsymbol{\theta}_m$ individually, we again arrive at equations (6.1) and (6.2).

In this work, we propose a structure learning algorithm called the SET-SPN (Search, Expand and Tune - SPN) algorithm. We would also like this algorithm to work in an online fashion as it is more flexible in real-world situations. A simple solution to this problem is, to begin with a simple seed network, where all the input random variables are assumed to be independent.

## 6.1 Initial networks

We now implement two initial network structures that can be used to further expand upon. Examples of generative and discriminative initial networks are shown in Figure 6.1, where $N_z$ is the number of random variables being modelled. For the generative setting network the values $v_1$, $v_2$, ..., $v_{N_z}$ are leaf distributions over the input ($\mathbf{x}$) and output ($\mathbf{y}$) random variables. In the discriminative setting network $v_1$, $v_2$, ..., $v_{N_z-1}$ are leaf distributions over the input random variables $\mathbf{x}$.

For this discriminative network, we have $N_z - 1$ leaf distributions, represented in the bottom layer. The one discrete output random variable $y$ is represented by the top $K$ product nodes, with one node for every class label. Once an initial network is created, it can be iteratively expanded upon by making parts of the network more expressive. To do so, the SET-SPN algorithm tests different possible network expansions and records the likelihood change associated with each expansion.




*(a) Generative initial network where all the random variables are initially naively assumed to be independent.*

*(b) Discriminative initial network where the label random variable $y$ divides the network into $K$ classes. Each class represents a distribution over the input random variables ($\mathbf{x}$).*

*Figure 6.1: Initial networks for the generative and discriminative training settings. These networks serve as a starting point for the SET-SPN algorithm.*

The best expansion is then applied to the network. This process is repeated until the likelihood of the network reaches a satisfactory level. Note that to simplify the explanation of the SET-SPN algorithm, we describe its computations in the linear probability domain. However, it is important to remember that all the SPN computations are conducted in the logarithmic domain.

### 6.1.1 Network expansion

After an initial network is created, the network is iteratively increased in size by expanding product nodes. Each expansion makes the network slightly more expressive, as illustrated in Figure 6.2 where node $i$ and two of its children are cloned.

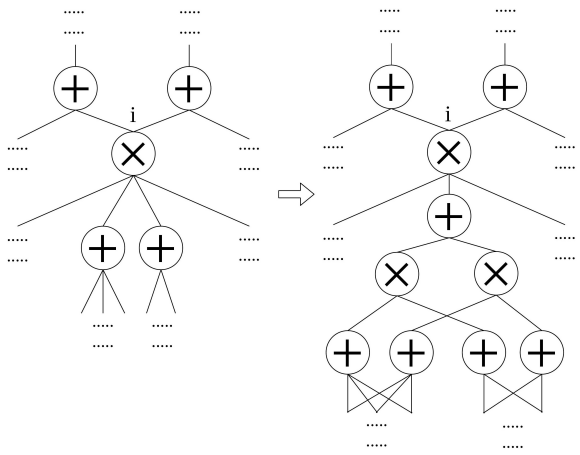



*Figure 6.2: Expansion of a product node $i$ in the network.*

Product nodes are targeted, as they represent local independence assumptions in the network. By correcting incorrect independence assumptions, the network can become better at representing the data. This MIXCLONES algorithm (Dennis & Ventura, 2015) creates two copies of a product node and two of its children. This expansion can be thought of as replacing two distributions, assumed to be independent, with a mixture with two components over those two distributions. The network thus becomes slightly more expressive after the expansion. This expansion process can then be repeated as training continues. It is important to note that we are cloning the original children nodes twice and changing the weights on only the clones. This is to prevent changes to other distributions that also rely on those original children nodes.

### 6.1.2 Equivalent networks

A challenge that remains is to optimally determine which nodes to expand and which parameters to assign to these newly expanded nodes. Finding the right node to expand in the discriminative training setting is especially difficult. Without an obvious solution to this problem, a direct naive method is investigated by randomly expanding different nodes with random parameters and recording the change in the likelihood of the data given the network. The expansion that increases the likelihood of the data the most is then picked to update the network. A typical problem with search methods like these is that they are of the computational complexity $\mathcal{O}(N_n N_d N_e)$, where $N_n$ represents the

number of nodes in the network, $N_d$ the number of data points, and $N_e$ the number of node-expansion tests. Two full passes through the network are needed to determine the effect of a network change on the likelihood of the data. Directly using search in this way is therefore too computationally expensive. The key insight we propose in this work is that the expansion search process can be decreased to $\mathcal{O}(N_n N_d + bN_e N_d)$, where $b$ is a real value normally close to the value 1, depending on how much search is conducted. This decrease in computational cost can be achieved by first calculating constants with complexity $\mathcal{O}(N_n N_d)$, which represent the equivalent network as seen by each node, before performing a search of complexity $\mathcal{O}(N_e N_d)$ to find the best expansion. To show how this is possible, we first denote the values for every leaf distribution for every data point $n$ as $v_{1n}, v_{2n}, \ldots, v_{Ln}$, where $L$ is the number of current leaf distributions in the network. We also denote the output of the current node we are interested in as $p_{in}$. An SPN can now be written in equation form, where the output $p_{rn}$ of the root node $r$ is equal to a first-degree polynomial operating on $v_{1n}, v_{2n}, \ldots, v_{Ln}$ and $p_{in}$, with only positive monomials and all exponent values equal to one. This is due to the network representing a valid probability distribution where product nodes have disjointed scopes for every child. A first-degree polynomial can be re-arranged into the form

$$p_{rn} = g_{in} p_{in} + c_{in}, \tag{6.4}$$

where $g_{in}$ and $c_{in}$ are the results of operations on the leaf distributions $v_{1n}, v_{2n}, \ldots, v_{Ln}$. The values for $g_{in}$ and $c_{in}$ remain constant for changes in $p_{in}$ if no other node's distribution changes. Therefore, if we can calculate the values for $g_{in}$ and $c_{in}$ we can effectively predict the network's output for changes in $p_{in}$, without needing to re-evaluate the network.

We can now also add regularisation to the network so that it does not become too large. To do this we add a regularisation term

$$\text{reg} = \frac{1}{1 + \lambda N_n}, \tag{6.5}$$

where $N_n$ indicates the number of nodes in the network. The value $\lambda$ is a regularisation term that indicates how much the number of nodes in a network affects the likelihood of the network. A $\lambda$ value of zero indicates no regularisation, while a larger $\lambda$ indicates that the optimiser should try to keep the network small. We also specify a maximum number

of nodes a network can have, e.g. 100 000. A prior probability can now be specified using

$$p(m) = \frac{\text{reg}}{\sum_{N_n} \text{reg}}. \tag{6.6}$$

In equation (6.6) we are summing over all possible values for $N_n$, from 1 to a maximum value e.g. 100 000. Due to the term $\sum_{N_n} \text{reg}$ being constant for changes in $m$, we can effectively remove it. This is because it does not influence the result of the argmax operation. We can, therefore, rewrite equation (6.3) to form

$$\hat{m}, \hat{\boldsymbol{\theta}}_{\hat{m}} = \underset{m,\boldsymbol{\theta}_m}{\text{argmax}}\, p(m) \prod_n \frac{p_m(\mathbf{x}_n, \mathbf{y}_n|\boldsymbol{\theta}_m)}{Z(p_m(\mathbf{x}_n|\boldsymbol{\theta}_m), g)} = \underset{m,\boldsymbol{\theta}_m}{\text{argmax}} \frac{1}{1+\lambda N_n} \prod_n \frac{g1_{in}p_{in} + c1_{in}}{g2_{in}p_{in} + c2_{in}}, \tag{6.7}$$

where the values $g1_{in}$, $g2_{in}$, $c1_{in}$ and $c2_{in}$ are constant for changes in $p_{in}$. This equivalent network is illustrated in Figure 6.3, where $p_{in}$ is the output of the product node one wants to expand for data point $n$.

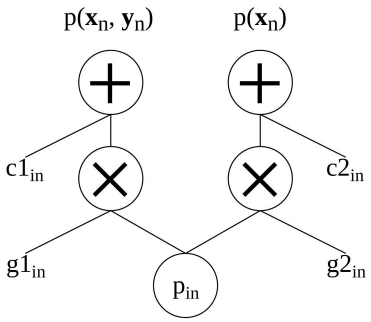



*Figure 6.3: Equivalent network seen by a network node $i$ in the discriminative setting. In the generative setting only $g1_{in}$ and $c1_{in}$ is needed to predict a normalised network's output.*

This essentially means that this equivalent network can accurately predict the change in likelihood for the entire network for any change to only one node. It is important to note that the same value $p_{in}$ is used in the numerator and the denominator. In the discriminative case, this is true for the specific initial network illustrated in Figure 6.1 (b). For discriminative training with more than one random variable in $\mathbf{y}$, different values need to be calculated for the numerator and denominator.

When a node is expanded using the MIXCLONES algorithm, only the network in Figure 6.3 now needs to be recalculated to find the likelihood change of the entire new

network. If one can now calculate these gradient ($g$) and addition ($c$) constants efficiently, node expansion tests can be conducted much quicker.

We now investigate how to efficiently calculate the $g1_{in}$, $g2_{in}$, $c1_{in}$ and $c2_{in}$ values for each node. By performing a joint forward pass, $p(\mathbf{x}_n, \mathbf{y}_n|\boldsymbol{\theta}_m, m)$ , and then a backward pass, the values for $g1_{in}$ and $c1_{in}$ can be calculated. The values $g2_{in}$ and $c2_{in}$ can equivalently be calculated by performing a marginal forward pass, $p(\mathbf{x}_n|\boldsymbol{\theta}_m, m)$, with a backward pass through the network after that. To calculate the gradient values, $g1_{in}$ and $g2_{in}$, the backpropagation algorithm is used, as described in Section 5.2.1.1. The backpropagation algorithm allows the gradients to be calculated in complexity time of $\mathcal{O}(N_n N_d)$. The challenge that remains is calculating the addition constants, $c1_{in}$ and $c2_{in}$, for every node in an efficient manner. The method used in this work also relies on backpropagation. If a child only has one parent, the addition constant can be calculated directly in the backward pass using

$$c_{in} = \begin{cases} c_{ln} + \sum_{j \in \text{children}(l) \sim i} p_{jn} w_{lj} & l \text{ is a sum node} \\ c_{ln} & l \text{ is a product node} \end{cases}, \tag{6.8}$$

where $l$ is the parent of node $i$, and $c$ can be substituted for $c1$ or $c2$. Here children$(l) \sim i$ is all the indices of the children of node $l$, excluding the child at index $i$. The $c1$ and $c2$ values for the root node are equal to zero. All the other $c$ values can be calculated recursively from the root node.

If a different initial network than that proposed in Figure 6.1 is used, then there might be non-leaf nodes with more than one parent. An exact algorithm is developed to calculate the constant values by first calculating the $g$ values for a node. The $c$ values can then be calculated using

$$c_{in} = p_{rn} - p_{in} g_{in}, \tag{6.9}$$

where $p_{rn}$ is the output of the root node at data point $n$, and $p_{in}$ is the initial pre-expansion output for node $i$ at data point $n$. This method is accurate in general but can have numerical instabilities as one is subtracting values, which may cause precision loss. If precision loss starts to occur, a computational graph can be created of all the additions that need to be performed to calculate $c1_{in}$ and $c2_{in}$ for nodes with more than one parent. The next step is to calculate the most efficient way to compute these constants using dynamic programming. Just adding numbers is numerically more stable than needing to add and subtract values, but might need more calculations. The final

number of calculations, to calculate $c1_{in}$ and $c2_{in}$, is on average still of the complexity $\mathcal{O}(N_d N_n)$.

### 6.1.3 Network search

Once the gradient and addition constants are calculated, random nodes can now be expanded and tested. Product nodes with more than one child are added to a list. Random samples are then taken from this list with random children and weights being chosen. The probability of choosing a product node is weighted by the number of children that the node has. This is implemented to make it more likely to choose product nodes with more children. Random expansions are then made and the likelihood after each expansion is calculated using that node's equivalent network. Similarly, random sum nodes are selected a certain percentage of the time, e.g. 10% of the time, and a random child node removed and therefore tries to compress the network slightly. The likelihood of this compressed network is then tested. Due to node regularisation, the compressed network can possibly be slightly more likely that other expansion candidates. After a certain number of iterations, the network selects the expansion or compression with the highest increase in likelihood to update the network. Some additional parameter tuning is also performed on the weights of the best expansion or compression to further increase the likelihood of the data, as discussed in the next section. This process is repeated until the likelihood of the network becomes satisfactory.

### 6.1.4 Weight fine tuning

After the best node to expand or compress has been found, the next step is to fine-tune the weights in this new node structure before it is added to the network. Therefore, we tune the weights of either the five sum nodes in the case of expansion or the one sum node in the case of compression. To tune these weights we investigate using standard first-order gradient descent, which was found to work best over Newton Raphson, and Expectation Maximisation. First-order gradient descent can also easily be implemented in the Python library PyTorch, which was used in our experiments.

By using equation (6.7), the likelihood of the network after the expansion or compression can be predicted. To achieve this, a forward pass through the equivalent network is first computed. Backpropagation is used to calculate gradients of the network's loss

with respect to each weight, and then the Adam optimiser is used to update these weight values. Once the weights have been updated for a certain number of iterations, the new network substructure, with updated weights, can be added to the network. The initial learning rate of the parameter-tuning algorithm is dynamically chosen to prevent nodes in earlier layers in the network not updating their parameters by a sufficient amount due to small gradient values. The learning rate is set to a small value and increased by a factor of 10 after each iteration until the weights start to change in value or a maximum learning rate value is reached. The iteration with the highest likelihood is used as the final expansion or compression parameters. We further experiment with the weight tuning capabilities of the SET-SPN algorithm in Section 7.2.2.1 and investigate the convergence of both algorithms, in the generative setting, on a toy problem in Appendix C.

## 6.2 Compressing an SPN

One problem with SPN structure learning algorithms, in general, is that they create highly tree-like models with duplicate network structures that exactly or approximately represent the same distributions. This means that the network creates wasteful duplications of similar structures. The efficiency of structure search algorithms also starts to diminish as the network grows with respect to the number of product nodes. To combat this problem, we developed a compression algorithm that can compress a network in time linear to the size of the network. Compression algorithms also help to regularise a network as they reduce the network's capabilities to model noise in the data. The requirements of this compression algorithm are that it should not decrease the accuracy of the network, while also having fast execution times. With these two constraints, we investigated how to reduce the number of network nodes and parameters as much as possible.

We subsequently designed a new compression algorithm called Compress-SPN. This compression algorithm performs two tasks, while still only taking time linear to the size of the network. One easy compression strategy is to delete sum-node weights that are close to zero. Weights that are close to zero indicate that the sum node does not consider that child's output as important at all. Therefore, Compress-SPN deletes these weights without any significant loss to the network's likelihood score. An additional step can then be taken to find nodes that do not have any remaining parent nodes. If a node does not have any remaining connections to a parent node, that node is redundant and can be

removed. This whole compression operation can be executed by performing one linear pass through the network, which makes it fast compared to other compression algorithms.

The second part of the Compress-SPN algorithm is to try to merge nodes. This second method of compressing a network, employed in Rahman & Gogate (2016), is used to find nodes with similar distributions. If two nodes have the same, or approximately the same, distributions, one of the nodes can effectively be deleted and its parents redirected to the other node. Therefore, if one compares every node's distribution to every other node's distribution, the network can be further compressed. There however lies an obvious problem in comparing every node with every other node, as this has a time complexity of $\mathcal{O}(N_n^2)$ to execute, where $N_n$ is the number of nodes in the network. We can try to speed up this process by only comparing nodes with the same subset of random variables, but this still requires a non-linear number of calculations in the size of the network. This would still not be fast enough for real-time use between structure expansions in SET-SPN, for example.

To improve the speed of this node-testing method, we propose using a hash table in the Compress-SPN algorithm. Hash tables have an average lookup time of $\mathcal{O}(1)$. Thus if we could do one lookup and insertion for every node, the compression algorithm can be implemented in time linear to the size of the network. We do this by saving the computation that every node effectively represents as a string hash. This is a somewhat crude, approximate method of comparing distributions, but it allows for comparisons to be performed in a much faster manner.

To do this comparison, we first round all the weights to a specific number of decimal points. Then we generate a string representing that node's computation. The string representation of the root node's computation in Figure 6.4 thus looks like $0.5(0.4z20 + 0.6z21)(0.3z10 + 0.7z11) + 0.5(z11)(z22)$, where $zij$ represents $\mathbb{1}_j(z_i)$.

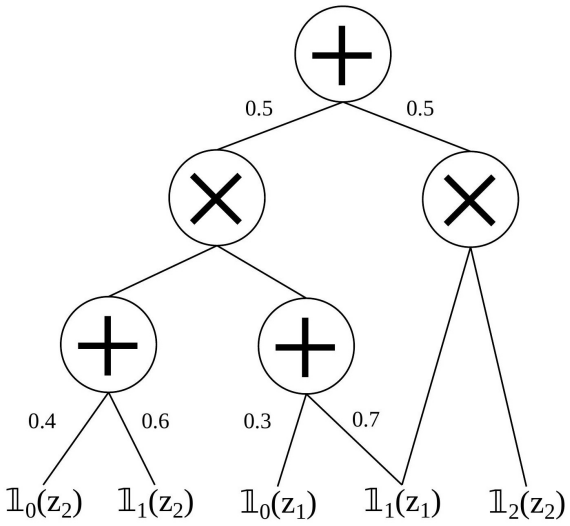



*Figure 6.4: An example of an SPN configuration.*

We can now compare these strings in a hash table and find matches between nodes in time linear to the size of the network. The Compress-SPN algorithm works row by row until all the nodes in the network have been tested. Algorithm 3 describes how the Compress-SPN algorithm works.

Algorithm 3 is designed to be executable in time linear to the size of the network. This fast execution time allows this algorithm to be used in between other learning algorithms' iteration steps to keep the network node count as low as possible.

**Algorithm 3** Compress-SPN algorithm used to compress an SPN in time linear to the size of the network.

**Data:** Network structure, cutLimit, roundLimit.

**Result:** Compressed network.

Initialise empty hashTable.

**for** $r$ in number of rows **do**
  **for** $c$ in number of columns **do**
    node = network$[r][c]$
    **if** node is sum **then**
      **for** weight in weights of node **do**
        **if** weight $<$ cutLimit **then**
          Remove weight.
      **end**
    nodeHash = generate hash using children, current node and roundLimit.
    **if** nodeHash in hashTable **then**
      Redirect node's parents to hashTable[nodeHash].
      Remove node.
    **else**
      hashTable[nodeHash] = node
    **end**
  **end**
**end**

Garbage collect all nodes without parents.

## 6.3 Conclusion

In this chapter, we investigated deriving a complete SPN directly from data using structure learning. We derived a new structure learning algorithm called SET-SPN. This algorithm randomly searches a network for substructures that do not fit the data well and should be expanded upon or compressed. The weights of the nodes then get updated using gradient descent, before getting incorporated into the network. Gradient descent was chosen to tune the network's weights as it works in all common training settings (discriminative, generative, discrete and continues variables) and achieves high accuracies compared to other methods. Lastly, we also created a new compression algorithm, called Compress-SPN, that can compress an entire SPN in between structural expan-

sions or compressions, performed by the SET-SPN algorithm. Using this Compress-SPN algorithm, therefore, helps remove redundant nodes from the network. There are, therefore, two possible compressions that can be done on the SPN, one from the SET-SPN compression tests and one from the Compress-SPN algorithm. This reduction in nodes also means that the SET-SPN can do more search per node and, therefore, find better expansion or compression candidates.

# Chapter 7

# Analysis of results

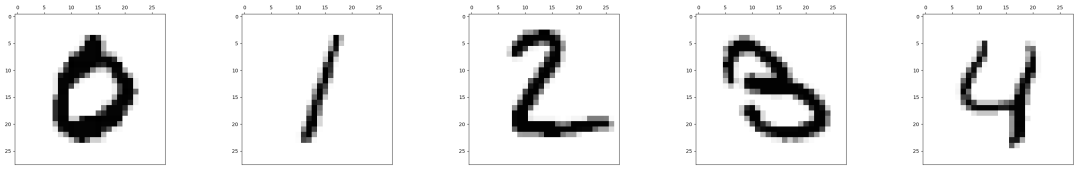
In this chapter, we evaluate different training and inference capabilities of sum product networks. We start by presenting the datasets used to evaluate the different algorithms. We then investigate the capabilities of the Compress-SPN and weight fine tuning algorithms and compare different SPN learning algorithms' performance with each other. Lastly, we compare the SPN's discriminative accuracies with other machine learning algorithms. In the second part of our experiments, as presented in Section 7.2.5, we also investigate an SPN's ability to work with partial observations.

## 7.1 The datasets

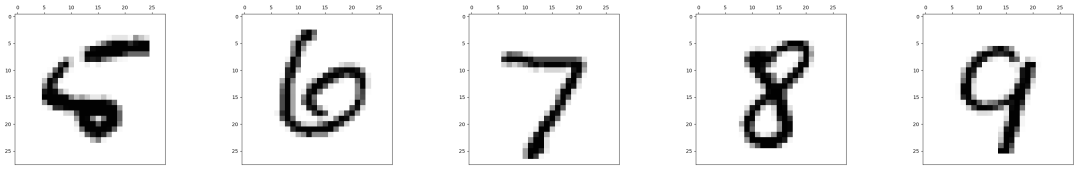
In our experiments we use two datasets, namely the MNIST and Fashion MNIST datasets. The MNIST dataset was chosen as it is a benchmark dataset with published results for almost all the SPN learning algorithms. Therefore, our results can easily be compared to the results obtained by others. We also include the Fashion MNIST dataset as it is more difficult to achieve high classification accuracies on than the vanilla MNIST dataset. This dataset also has the same input and output dimensions as the MNIST dataset and therefore all the training algorithms can easily switch between the two datasets.

### 7.1.1 MNIST

As briefly described in Section 4.7.2.2, the MNIST dataset[1] consists of 60 000 training data points and 10 000 test data points. Each data point represents a two-dimensional image, of $28 \times 28$ greyscale values, and a corresponding digit label between the values of 0 and 9. The network is tasked with classifying these digits for every input image. These 10 types of digits in the MNIST dataset are represented in Table 7.1.



*(a) Zero* *(b) One* *(c) Two* *(d) Three* *(e) Four*



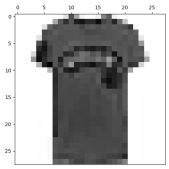
*(f) Five* *(g) Six* *(h) Seven* *(i) Eight* *(j) Nine*

*Figure 7.1: Examples of the 10 different classes/digits present in the MNIST dataset.*

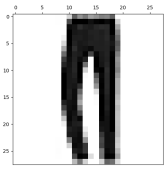
### 7.1.2 Fashion MNIST

The Fashion MNIST dataset[2] also consists of 60 000 training data points and 10 000 test data points. Each data point represents a two-dimensional image with $28 \times 28$ greyscale values and a corresponding label. These images represent one of ten types of clothing, labelled 0-9. Figure 7.2 illustrates the ten types of clothing presented in the Fashion MNIST dataset.

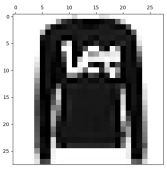
[1] MNIST dataset hosted on: http://yann.lecun.com/exdb/mnist/
[2] Fashion MNIST dataset hosted on: http://github.com/zalandoresearch/fashion-mnist



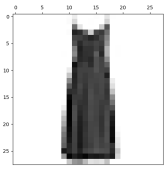
*(a) T-shirt/top (0)*



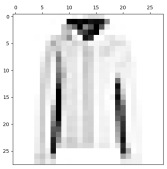
*(b) Trouser (1)*



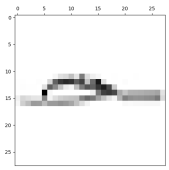
*(c) Pullover (2)*



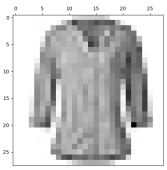
*(d) Dress (3)*



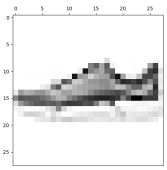
*(e) Coat (4)*



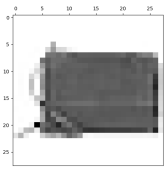
*(f) Sandal (5)*



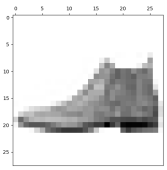
*(g) Shirt (6)*



*(h) Sneaker (7)*



*(i) Bag (8)*



*(j) Ankle boot (9)*

*Figure 7.2: Examples of the ten different classes present in the Fashion MNIST dataset.*

With these datasets in place, we can now start evaluating different learning algorithms. We start by evaluating the Compress-SPN algorithm.

## 7.2 Training of SPNs

### 7.2.1 Investigation on compression

In this section, we investigate the effectiveness of the compression algorithm proposed in Section 6.2. We would like to determine how much each component of the Compress-SPN algorithm contributes to the overall compression of an SPN. It is important that the compression algorithm does not reduce the classification accuracies of the networks. We would, therefore, also like to test that after the network has been compressed it is still as accurate as before its compression. We would also like to compare the final parameter compression percentage of our algorithm with the published results on the tSPN algorithm.

We evaluate the Compress-SPN algorithm by testing it on three different networks trained using three different learning algorithms. These networks are trained on the MNIST dataset in the discriminative setting. The learning algorithms we use include the RAT-SPN, DSPN-SVD and SET-SPN algorithms. For the SET-SPN algorithm, we do not compress the network while the network is training. We now investigate how well

different components of the Compress-SPN algorithm does individually and how well they do combined. This result can be seen in Table 7.1.

*Table 7.1: Average compression results, using the Compress-SPN algorithm applied on three trained uncompressed networks. These networks were trained using the RAT-SPN, DSPN-SVD and SET-SPN algorithms. For this experiment, the SET-SPN algorithm is not compressed in between expansions. In the 'Zero weight' category only weights with values close to zero were deleted. In the 'Duplicate structure' category the compression algorithm only searches for duplicate structures to delete. In the final 'Combined' category both previously mentioned algorithms are used. However, due to there existing duplicate structures with near-zero weights in, the final compression result is less than the sum of the individual algorithms.*

| Algorithm | Accuracy loss (%) | Node compress (%) | Weight compress (%) |
|---|---|---|---|
| Zero weight | 0.0 | 10.1 | 20.1 |
| Duplicate structure | -0.1 | 50.1 | 42.3 |
| Combined | -0.1 | 55.4 | 51.4 |

The accuracy loss indicates what the average classification accuracy on the MNIST test set after compression is, subtracted from the accuracy before compression. The results in Table 7.1 indicate that the networks' testing accuracies did not deteriorate, and actually increased by 0.1% after compression was performed. There was never a network that achieved a lower testing accuracy after compression than before. This indicates that compressing the networks helps remove some structures that model noise in the data, and therefore slightly reduces overfitting in the networks. Both the 'Zero weight' and 'Duplicate structure' algorithms have a significant enough contribution to the final compression score that both routines should be included in the final Compress-SPN algorithm. The Compress-SPN took on average 0.95 seconds to compress a network for every 100 000 parameters. This allowed the algorithm to always take less than 3 seconds to compress any of the networks, which is fast enough to be applied in the SET-SPN learning algorithm. As seen in Table 7.1, the Compress-SPN algorithm also removes, on average, around half of the network's nodes and parameters with negligible loss in accuracy. This compression especially helps online structure learning algorithms, like the SET-SPN algorithm, as it reduces unnecessary structural tests. The Compress-SPN

algorithm also has an execution time that scales linearly to the size of the network. Therefore, the algorithm can be applied regularly in-between expansions in structure learning algorithms.

The Compress-SPN algorithm, however, achieves less network reduction than the reported results for the tSPN, proposed in Ko *et al.* (2018), and described in Section 2.3.3. In Ko *et al.* (2018), they reported compression capabilities of upwards of 90%. The current downside to this tSPN compression algorithm is that it creates a tensor network, which cannot be further used by structure learning algorithms. This is due to the tSPN usually not being a valid SPN, but still a valid probabilistic model. Therefore, determining which expansions are legal in this compressed probabilistic network is difficult, which complicates learning. The tSPN compression algorithm is estimated to take considerably longer than the compression algorithm proposed in this work, although no specific training times were found in Ko *et al.* (2018). It is, therefore, recommended to use the Compress-SPN algorithm in between online structure learning iterations and then apply the more powerful tSPN compression algorithm as a final step, after training.

### 7.2.2 Evaluation of SET-SPN

#### 7.2.2.1 Weight tuning on MNIST

The SET-SPN algorithm consists of three distinct steps, as indicated by its name. These steps are Searching, Expanding and Tuning (SET). The searching step involves randomly trying different network expansions and finding the one that produces the best likelihood increase. In the second expansion step, a new expansion is incorporated into the network, although compression is also possible. The final tuning step involves tuning the weights of the new nodes in the network.

In this experiment, we would like to determine which parameter learning algorithm works best in tuning weights for the SET-SPN algorithm. We investigate three different methods of tuning the parameters of the expanded or compressed structures in the network. The methods we evaluate are Expectation Maximisation (EM) and two gradient descent methods. The EM algorithm is a generative algorithm, but has fast convergence times. It is therefore of interest to see if this fast convergence time compensates for the fact that this algorithm is a generative algorithm applied in a discriminative training setting. For the gradient descent methods, we use first-order gradient methods. A

second-order gradient descent (Newton Raphson) method was also implemented in this work, but later abandoned due to the Hessian matrix regularly becoming non-invertible, and thus the computation could not be completed. This means that in some cases an update step could not be calculated using this Newton Raphson method. We, therefore, opted to use a more stable first-order gradient descent method. We also investigate using a more advanced first-order gradient descent optimiser called the Adam optimiser (Kingma & Ba, 2015). This optimiser has proven quite successful in training deep neural networks and can possibly achieve the same results in SPNs.

We now investigate how all three algorithms fair on the real-world MNIST dataset in the discriminative setting. We adapt equation (6.7) to derive the likelihood function we want to maximise as

$$\hat{m}, \hat{\boldsymbol{\theta}}_{\hat{m}} = \underset{m,\boldsymbol{\theta}_m}{\operatorname{argmax}} \frac{1}{1+\lambda N_n} \prod_n \frac{g1_{in}p_{in} + c1_{in}}{g2_{in}p_{in} + c2_{in}}, \tag{7.1}$$

where the values $g1_{in}$, $g2_{in}$, $c1_{in}$ and $c2_{in}$ are constant for changes in $p_{in}$. Here $n$ and $i$, respectively, represent the data point index and node index we are currently interested in. The value $N_n$ represents the number of nodes in the network and $\lambda$ is a real number that indicates how much the SET-SPN should focus on keeping the number of nodes in the network low. Here $\hat{m}$ and $\hat{\boldsymbol{\theta}}_{\hat{m}}$ represent the optimal model and optimal parameters for that model. In this experiment we apply the full SET-SPN algorithm, but with different weight tuning algorithms. Here $p_{in}$ again represents the probability output of the node that we want to expand. We start by trying to visualise how the MNIST data looks over two random variables. We, therefore, construct a random SPN and investigate expanding a random product node in that network. The data points over two pixel random variables, in the MNIST training dataset, are plotted in Figure 7.3.

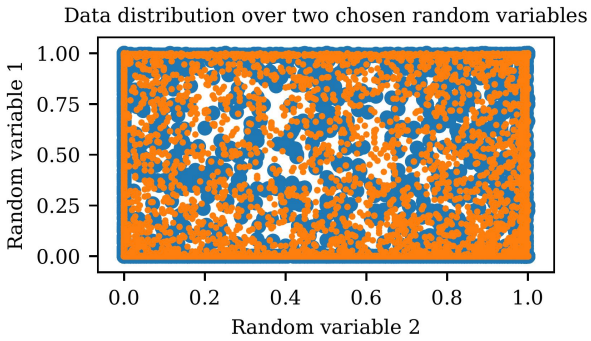



*Figure 7.3: Data distribution over two pixels in the MNIST training dataset, as seen by a specific node. The points that are estimated, using the gradients, that this node should represent are indicated in blue, while orange indicates points that should not be represented by this node.*

The gradient of the loss of the network with respect to each node indicates if that node should have a high or low probability output for a specific data point. Hence, gradients can be used to estimate if a node should or should not capture a given data point for the current update step. Because we are working in the discriminative setting, nodes are tasked with representing some of the data points, while also explicitly not representing some other data points. The network, therefore, does not just have to represent the data points but has to be able to discriminate between the classes. The blue and orange indicate if this local distribution should represent (indicated in blue) or explicitly not represent (indicated in orange) these points. The gradient descent algorithm can naturally handle this negatively valued data points along with the positive data points. However, the EM algorithm, as used in SPNs, is a generative algorithm and therefore only focusses on the positive weighted points.

We now test 3 different parameter learning algorithms in the SET-SPN algorithm and run them until they reach their maximum validation accuracies. The results are presented in Table 7.2.

The result seems to show that the Adam optimiser performs the best on average. Through evaluating the training of the EM algorithm it seems that this algorithm has fast training times but with lower accuracies in the discriminative setting, as it only works with positive data points.

*Table 7.2: Training times and classifications accuracies of the SET-SPN algorithm for different parameter learning algorithms, on the MNIST test dataset. Each parameter learning algorithm was tested twice, using the SET-SPN algorithm, and the average result is presented. The algorithms were allowed to run until they reached their maximum validation classification accuracy on the MNIST training set. This validation set was made up of 25% of the original MNIST training data.*

| Algorithm | Train time (hours) | Test set accuracy |
|---|---|---|
| EM | 85 | 90.9% |
| Standard gradient descent | 121 | 94.2% |
| Adam optimiser | 94 | 98.1% |

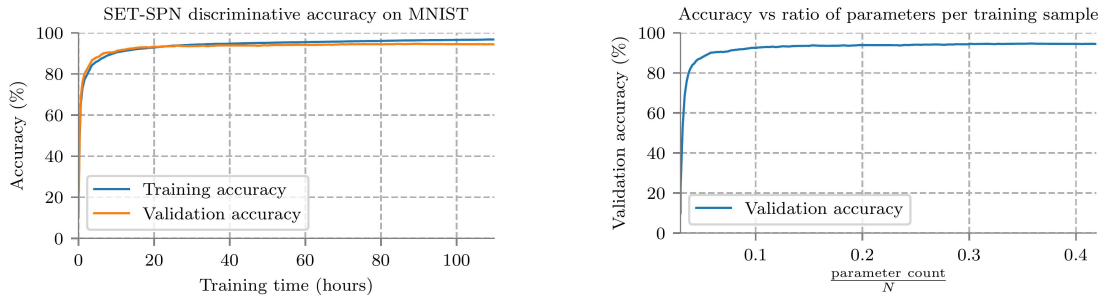
Gradient descent, however, excels with negative and positive examples as it is directly trying to minimise a loss function. We conclude that the Adam optimiser seems to work the best for tuning the weights of expanded nodes in a network.

#### 7.2.2.2 Capacity of SET-SPN

We now investigate the capacity and training times of the SET-SPN algorithm. In this experiment, we would like to determine if the SET-SPN algorithm can completely represent the MNIST training dataset. Therefore, we would like to see if, given enough time, the SET-SPN algorithm can overfit on the MNIST training dataset. This is done to prove that the SET-SPN algorithm is at least expressive enough to represent the MNIST dataset.

For this experiment, we do not add the node regulation term, specified in equation (6.5), to the SET-SPN algorithm as we are not interested in the algorithm's ability to generalise. We also only allow the SET-SPN algorithm to test expansion nodes and not compressions as we are not interested in compact networks. After each expansion, we perform gradient descent on the expanded nodes' weights for 1 000 iterations. The discriminative network in Figure 6.1 (b) was used as an initial network. The entire training set was used in this experiment to train on. For the validation set we used the 10 000 digits of the MNIST test set. The training and validation accuracies for the first 110 hours are shown in Figure 7.4 (a) with a maximum validation accuracy occurring

around the 94-hour range. Figure 7.4 (b) shows the validation accuracy against the ratio of the number of parameters divided by the number of training data points.




*(a) Accuracy* *(b) Parameter count ratio*

*Figure 7.4: Training accuracy and parameter count of a SET-SPN trained using the MNIST training dataset. Note that this accuracy is lower than the result we later obtained in Section 7.2.3, due to no node regularisation being included in the network. During the first 10 hours of training, the network severely underfits to the data. This underfit model actually, coincidently, has a validation accuracy higher than the training accuracy for a small period of time, due to the small number of nodes in the initial network.*

The SET-SPN algorithm was able to achieve a 100% overfit on the 60 000 digits of the MNIST training set after 171 hours. Therefore, the SET-SPN is expressive enough to represent large datasets. An interesting observation is that the network began to overfit on a parameter data ratio of about 0.35. Therefore, the network starts overfitting when the number of parameters is just above a third of the number of training samples. This shows that the network can become fairly large before it begins to overfit. This metric also provides another method for estimating when to stop training to avoid overfitting.

### 7.2.3 Comparative study

In this next section, we compare popular SPN training algorithms with one another in the generative and discriminative settings. The reason we do this is two-fold. Firstly, we would like to determine how well the SET-SPN algorithm does in comparison to the other popular learning algorithms for SPNs. Secondly, we would also like to determine the best learning algorithms to use in the discriminative and generative settings.

We evaluate these algorithms on the popular MNIST dataset. In Section 7.2.4, we also evaluate SPNs and other models on the Fashion MNIST dataset, which is considered a more difficult classification dataset than MNIST. The reason we opt to test our implementations of the SPN algorithms on the MNIST dataset is because almost all the discriminative training algorithms have results using this dataset. Therefore, we can easily compare our results to the results obtained in the original papers. For this comparative study, we evaluate the different structure and parameter learning algorithms against each other. We next discuss the settings used for all the different learning algorithms that are tested.

#### 7.2.3.1 Training algorithm configurations

In this experiment, we test 5 different learning algorithms, from which 4 are structure learning algorithms and one is a parameter learning algorithm. These algorithms are the SET-SPN, RAT-SPN, SPN-SVD, DSPN-SVD and LearnSPN algorithm. While we discuss other algorithms, we do not explicitly evaluate SPN architectures that are designed for specific data types, like Dynamic SPNs and Convolution SPNs. In this section, we, therefore, focus on evaluating general learning algorithms for SPNs.

We first describe the settings used in the anytime structure learners, namely SET-SPN and RAT-SPN. They are anytime algorithms because they can be stopped at any reasonable point in time and a valid SPN can be retrieved. The longer the algorithm is trained for usually means the more accurate the network is in representing the data. For the SET-SPN algorithm, there are 10 000 expansion tests done before an expansion is chosen. We chose the node regularisation term $\lambda = 2 \times 10^{-4}$. The SET-SPN algorithm is also instructed to select a candidate sum node, 10% of the time, and deletes one of its children, thus compressing the network. Because there is node regularisation, sometimes this compression candidate is chosen as the best change to incorporate into the network. The weight fine-tuning is then conducted for 1 000 iterations on the final candidate node structure. For the RAT-SPN algorithm an initial network with hyperparameters $D = 3$, $S = 5$, $R = 25$ was used, where $2D$ is the network depth. The number of sum nodes in the non-leaf and non-root regions is represented by $S$. The number of root sum nodes is represented by $R$. This results in a network with 221 010 parameters. We chose these hyperparameters to create a large enough network, while also having fast enough

training times. We chose the parameter regularisation term $\lambda = 0.01$ for the RAT-SPN algorithm. For both the SET-SPN and RAT-SPN algorithms we used early stopping on the validation set (25% of training data) to determine when to stop training the networks. This means that when the validation accuracy started to decrease, by a sufficient amount, the training algorithm is stopped. We use 15 000 MNIST images as validation data. The remaining 45 000 training images are therefore used by the models to directly train from. For all the algorithms used we allow for a maximum training time of 190 hours. For parameter learning on the RAT-SPN and SET-SPN node expansions, we use the Adam optimiser, available in the PyTorch library. For hardware specifications refer to Appendix B.

All 60 000 training examples in the MNIST training set are used for the LearnSPN, SVD-SPN and DSVD-SPN algorithms, as no validation is needed. In the case of the LearnSPN algorithm, we use EM clustering to find clusters within the data instances. The LearnSPN algorithm is a recursive algorithm that iterates between searching for independence (creating product nodes) and clustering data (creating sum nodes). In our implementation of the LearnSPN algorithm, we use the G-test of pairwise independence to determine if random variables are independent over the subset of data provided. The G-test of pairwise independence is one of the standard methods used to estimate if two discrete random variables are independent of one another. A G-test significance value ($p$) of 0.0015 is used as recommended in Gens & Domingos (2013). Therefore, if the G-test significance value is below 0.0015, the variables are considered to be independent. Because the LearnSPN algorithm only works with discrete random variables, the data needs to be made binary to allow the G-test to test for independence, and then the data gets transformed back to its original form. In our LearnSPN implementation, the data gets clustered using an EM algorithm on an increasing number of clusters ranging from two to five clusters, with an added penalty for the number of clusters used.

For the SPN-SVD and DSPN-SVD algorithms, we use the constraint $\gamma = 2$. The value $\gamma$ controls the penalty incurred as each matrix being considered deviates from being a rank-1 matrix. This value should be larger than one, but not too large so that no large enough approximately rank one sub-matrices (atoms) can be found. The value of 2 was found to work well. In the case of the DSPN-SVD algorithm we have another constant called $d$, which indicates the number of features that should be extracted. We set $d$ to be 25% of the number of input random variables. Therefore, in the case of MNIST and

Fashion MNIST we have $d = 28 \times 28 \times 0.25 = 196$. For the feature extraction part of the DSPN-SVD algorithm, we also use the G-test of pairwise independence to find the most correlated, binary converted, inputs to the output class variable.

#### 7.2.3.2 Discriminative training

We start by evaluating how well each SPN learning algorithm does in the discriminative setting. In the discriminative setting, the raw continuous domain MNIST images are provided to the algorithms. The algorithms are then trained using the MNIST training set and evaluated on their classification accuracy on the MNIST test set. To work with these continuous domain inputs the learning algorithms work from Gaussian leaf distributions. Due to the LearnSPN and SPN-SVD algorithms being generative algorithms, they are trained generatively. They are then tested based on their discriminative classification accuracies. The classification accuracy over time of each algorithm on the MNIST test set is presented in Figure 7.5.

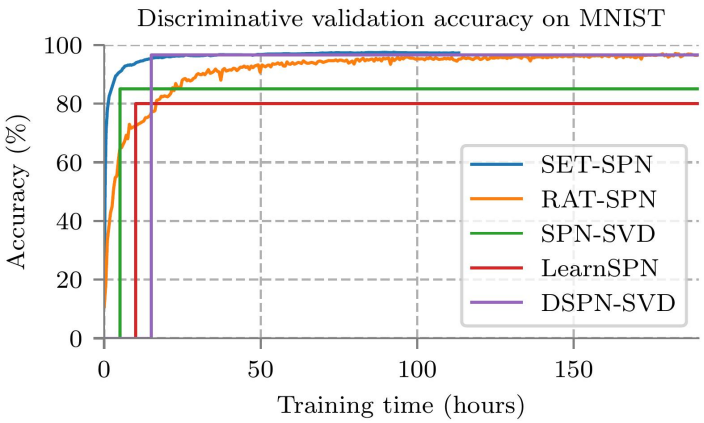



*Figure 7.5: Classification accuracies of 5 SPN learning algorithms on the MNIST test dataset, presented over time. Note that the LearnSPN, SPN-SVD and DSPN-SVD algorithms only construct an SPN after they have completed their fixed computation. This is indicated with an upward step on the graph at the time when the algorithm finished its computations.*

Figure 7.5 indicates that the SET-SPN algorithm achieves the highest discriminative classification accuracy, closely followed by the RAT-SPN and DSPN-SVD algorithms.

The SET-SPN algorithm stopped early as it reached its maximum validation accuracy at that point and achieved a classification accuracy, on the testing set, of 98.1%. The RAT-SPN and DSPN-SVD algorithm, respectively, achieves a test set classification accuracy of 97.7% and 97.6%. The results published in Peharz *et al.* (2018) for the RAT-SPN algorithm did however achieve a higher MNIST classification accuracy of 98.19%. It might be possible that they trained their RAT-SPN model for more than 190 hours before reaching the reported accuracy, or used multiple (or more powerful) GPUs to train their models. There have also been recent results published using a convolution SPN that achieved a classification accuracy on the MNIST test set of 99.19% (Van de Wolfshaar & Pronobis, 2019). The convolutional SPN can, however, only be used on spatially related input data. The SET-SPN algorithm, however, becomes a valuable tool when one considered the size of the network that was generated. The SET-SPN algorithm generated a network that only has 22 634 parameters. The RAT-SPN was trained using a network of size 221 010 parameters, and the DSPN-SVD having 252 773 parameters. Thus the SET-SPN achieves a higher classification accuracy with a model with around 10 times fewer parameters that the RAT-SPN and DSPN-SVD algorithms. We will evaluate the classification accuracy of a similarly sized RAT-SPN in Section 7.2.3.4. Another advantage of the SET-SPN algorithm is the fact that it works on continuous and discrete random variables in the generative and discriminative setting. No initial network structure is also needed, which means there are no bad initial structures generated as are with other parameter learning algorithms. We, therefore, recommend using the SET-SPN algorithm to initially generate a network for a dataset. Then after this SET-SPN is created the RAT-SPN parameter learning algorithm can be used to fine-tune the weights of the network, and therefore combine the best attributes of both algorithms.

#### 7.2.3.3 Generative training

We now evaluate how well each learning algorithm does in the generative setting. In this setting the algorithms are tasked with generatively modelling both the input pixels ($\mathbf{x}$) and output label ($y$). The log likelihood loss used for each model, on the MNIST test set, to determine the model's accurate is described by

$$\text{Likelihood loss} = -\sum_{n} \ln(p_m(\mathbf{y}_n, \mathbf{x}_n|\boldsymbol{\theta}_m)). \tag{7.2}$$

We exclude the DSPN-SVD algorithm as it is designed for the discriminative setting. We would like to provide a fair comparison for the LearnSPN and SPN-SVD algorithms as they are designed to achieve good results in the generative setting. In this generative setting, we therefore opt to discretise the pixel values in the MNIST dataset for all the algorithms. We do this by setting all the pixels as either 0 (white) or 1 (black). The SVD-SPN can work with continuous data, as is done in Section 7.2.3.2, but it was found to work better with discrete data. The results of the algorithms trained in this generative manner are presented in Figure 7.6.

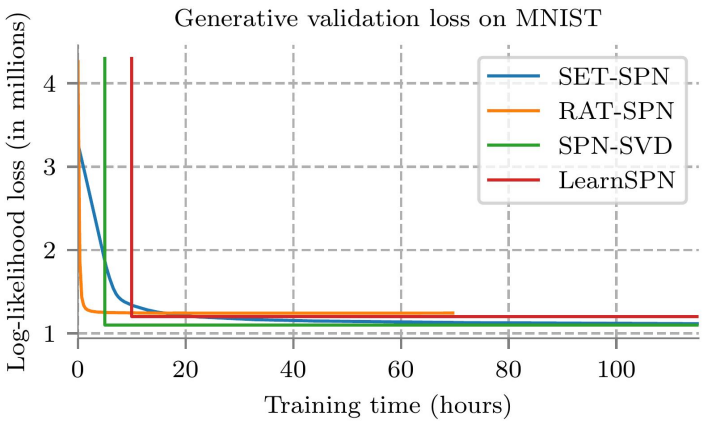



*Figure 7.6: Generative data log-likelihood losses of 4 algorithms trained in a generative manner on a binary input version of the MNIST dataset. Note that the LearnSPN and SPN-SVD algorithms only construct an SPN after it has completed its fixed computation. This is indicated with a downward step on the graph at the time when these algorithms finished their computations. Any value before this downward step is therefore irrelevant for the algorithm. All algorithms stopped before the maximum training time of 190 hours.*

As can be seen in Figure 7.6, the SPN-SVD algorithm achieves the best result with the lowest log-likelihood loss, and the SET-SPN algorithm achieves the second-best result. The SET-SPN algorithm again constructs a compact network, with 21 643 parameters, compared to the RAT-SPN of size 221 010 parameters. We now investigate how the RAT-SPN's accuracy changes when we use a network of similar size to the network generated by the SET-SPN algorithm. We discuss our findings on this smaller RAT-SPN next.

#### 7.2.3.4 Reduced RAT-SPN

Considering the results obtained in Figure 7.5 and Figure 7.6, one can see that the SET-SPN algorithm seems to perform slightly better than the RAT-SPN algorithm in our implementations. The RAT-SPN is, however, trained using a large initial network, which could potentially increase training times. We now investigate how the test-set accuracy of the RAT-SPN algorithm changes if we initialise a RAT-SPN with approximately the same size as the one generated by the SET-SPN algorithm. This is done to determine if we can achieve a better discriminative classification accuracy. We construct a RAT-SPN with hyperparameters $D = 3$, $S = 3$, $R = 5$. This results in a network that has 24 790 parameters compared to the 22 634 parameters of the SET-SPN. The final results of all the evaluated SPN learning algorithms, in the discriminative setting, are presented in Table 7.3.

*Table 7.3: Discriminative classification accuracies obtained from different learning algorithms on the MNIST dataset.*

| Algorithm | Training time (hours) | Parameters | Disc accuracy |
|---|---|---|---|
| SET-SPN | 98.1 | 22.7k | 98.1% |
| Large RAT-SPN | 190 | 221k | 97.7% |
| Small RAT-SPN | 190 | 24.8k | 82.4% |
| SPN-SVD | 5 | 83k | 86.2% |
| DSPN-SVD | 15 | 252k | 97.6% |
| LearnSPN | 10 | 127k | 80.4% |

In Table 7.3, we trained the smaller RAT-SPN for the same amount of time that the larger network was trained for. By looking at the results we can see that the smaller RAT-SPN achieves a lower test set accuracy than the larger RAT-SPN. One hypothesis as to why this smaller network has a reduced accuracy is due to how RAT-SPNs are generated. RAT-SPNs generate random substructures that might have configurations in them that do not fit the data well. These bad substructures are effectively ignored during training, while the good substructures are utilised. However, if the network is large there is a larger number of substructures to chose from and exploit. This allows the network to better model the data using the good substructures. To test this hypothesis, we evaluate

the number of near-zero connections there are in each network. We present the results in Table 7.4.

*Table 7.4: Results on pruning weights in a large RAT-SPN (221k parameters) and smaller RAT-SPN (24.8k parameters), which were both trained on the MNIST dataset. Weights are pruned if they are smaller than* $10^{-5}$*. This table provides the percentage of weights that could be pruned from the network. Both networks have a negligible loss in accuracy after pruning.*

| Network | Weight pruning % |
|---|---|
| Large RAT-SPN | 27.5% |
| Small RAT-SPN | 7.4% |

The results in Table 7.4 show that a large RAT-SPN disregards a larger portion of its internal nodes than the smaller RAT-SPN. This seems to support the claim that if a network has more structure to choose from it can disregard more bad structural components. This might be due to the network needing a minimum amount of nodes to be expressive enough to represent the data. Recently published research also seems to indicate that similar effects might be plaguing other types of neural networks (Frankle & Carbin, 2018) as well, where some parts of these randomly generated networks are essentially untrainable. Therefore, in the case of the RAT-SPN algorithm, some of these bad initial substructures might very well be untrainable for a given dataset.

Now that we have learning algorithms available to generate an SPN, we would like to compare the results SPNs obtain to those obtained by other models. SPNs need to have fairly high accuracies before it is worth considering their other attractive properties. We will provide this comparison in the next section.

### 7.2.4 Accuracies compared to other models

In this section, we investigate the accuracies of SPNs compared to some other machine learning algorithms. SPNs have a wide range of inference capabilities, but if they cannot model data well enough, compared to other models, this inference capabilities will not be accurate. We would, therefore, like to compare the discriminative accuracy of SPNs with other relevant models.

The models we compare the SPN algorithm to are standard feedforward neural networks, in the case of function approximators, and the Naive Bayes model and Gaussian Discriminant Analysis (GDA), in the case of probabilistic models. We opt to use a feedforward neural network, as it provides a more fair comparison to the feedforward SPN architecture, used in this work. Both the probabilistic models that are tested are traditionally generative models that can be used for discriminative classification. We, however, also train these probabilistic models directly in the discriminative setting using gradient descent. These two probabilistic models are tested as they can scale quite easily to large datasets like MNIST and Fashion MNIST, which are the two datasets we are evaluating the algorithms on. For both the probabilistic models, in the generative setting, we use Bayes' rule to generate a prediction as follows

$$p(y|\mathbf{x}) = \frac{p(\mathbf{x}|y)p(y)}{p(\mathbf{x})}. \tag{7.3}$$

We do not have to represent the probability values $p(\mathbf{x})$, as we can just normalise the probability values $p(\mathbf{x}|y)p(y)$ to calculate $p(y|\mathbf{x})$. Secondly, to boost the accuracies of these probabilistic models, we also directly train these models in the discriminative setting using gradient descent. In this setting we directly maximise $p(y|\mathbf{x})$ for both these models.

The Naive Bayes model is (naively) based on the assumption that

$$p(\mathbf{x}|y) = \prod_i p(x_i|y), \tag{7.4}$$

where the product iterates over all the observed random variables. In our experiment we evaluate using both an univariate Gaussian distribution and univariate Weibull distribution (Kumar & V, 2017) to represent $p(x_i|y)$. The Gaussian distribution is defined over an input continuous random variable space of $(-\infty, \infty)$. The adapted univariate Weibull distribution is also defined over an input continuous random variable space of $(-\infty, \infty)$, but can only output non-zero probability density values for the range $[t_y, \infty)$. We slightly adapt the original Weibull distribution by including an offset value, for each class $y$, represented by $t_y$. This value is important, because if this offset value was not included the distribution would always have output zero for an input value of zero. We can now express this adapted Weibull distribution mathematically using

$$p(x_i|y) = \begin{cases} \frac{k_y}{s_y}(\frac{x_i - t_y}{s_y})^{k_y - 1} e^{-(\frac{x_i - t_y}{s_y})^{k_y}} & x_i \geq t_y \\ 0 & x_i < t_y \end{cases},$$

where $y$ is the state value of the current label being modelled. The value $k_y$ is a continuous value ($k_y > 0$) and influences the shape of the distribution. The value $s_y$ is also a continuous value ($s_y > 0$) and influences the scale of the distribution. In this experiment we assign $1 < k_y < 4$ and $0.2 < s_y < 2$ to allow for stable gradient updates. The value $x_i$ represents the state value of the observed random variable at index $i$.

In GDA we assume that every category (label) in the output random variable ($y$) is represented by a multivariate Gaussian distribution which is defined over all the input random variables ($\mathbf{x}$). A multivariate Gaussian distribution, in the log domain, is defined as

$$\ln(p(\mathbf{x}|\boldsymbol{\mu}, \Sigma)) = -\frac{1}{2}\left[(N_Z - 1)\ln(2\pi) + \ln|\Sigma| + (\mathbf{x} - \boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x} - \boldsymbol{\mu})\right],$$

where $\mathbf{x}$ is the set of observed random variables the Gaussian is defined over. The values $\boldsymbol{\mu}$ and $\Sigma$ represent the mean and covariance (related to standard deviation) matrices. Here $N_Z$ represents the number of random variables we have. The inference random variable is not modelled by each GMM, therefore we subtract one from $N_Z$. The $T$ operator signifies that the matrix should be transpose. Lastly the $|\Sigma|$ and $\Sigma^{-1}$ operators indicate, respectively, that the determinant and inverse of the $\Sigma$ matrix should be taken.

If one simply trains a multivariate Gaussian over all the 784 input random variables, the $|\Sigma|$ value starts to approach zero, which means the functions start to output values approaching infinity. This is due to some random variables having zero variance (or standard deviation), which produces infinitely spiking Gaussians. To combat this problem we first find the most correlated random variables using the G-test of pairwise independence, on a discrete version of the MNIST and Fashion MNIST datasets where all input pixels are rounded to either zero or one. This G-test is performed to find the random variables that are most correlated with each class. A multivariate Gaussian is then constructed on the subset of input random variables and we assume that the class is independent of the other random variables. We also add a small amount of noise to the data to make sure no random variable has exactly zero variance (standard deviation). In Table 7.5 we present the results of the different learning algorithms on the MNIST and Fashion MNIST datasets. In the results, we also display the accuracies obtained for each probabilistic model, trained directly in the discriminative setting using gradient descent. In the discriminative setting we still use the same model setup, but directly optimise the model's parameters to maximise the discriminative accuracy of that model. These

discriminatively trained versions are indicated with a 'Disc' text next to the algorithm name.

*Table 7.5: Classification accuracies, obtained from different learning algorithms, on the MNIST and Fashion MNIST test datasets. Here 'Disc' indicates that this probabilistic model was directly trained in the discriminative setting using gradient descent. The 'Gen' keyword indicates that the network's parameters were estimated generatively, as is usually done for the probabilistic model.*

| Algorithm | MNIST accuracy | Fashion MNIST accuracy |
|---|---|---|
| SPN | 98.1% | 89.1% |
| DNN | 98.8% | 90.2% |
| GDA (Gen) | 86.9% | 72.7% |
| GDA (Disc) | 93.2% | 83.2% |
| Naive Bayes (Gaussian, Gen) | 73.7% | 65.3% |
| Naive Bayes (Gaussian, Disc) | 91.3% | 80.1% |
| Naive Bayes (Weibull, Gen) | 77.4% | 70.8% |
| Naive Bayes (Weibull, Disc) | 92.0% | 84.3% |

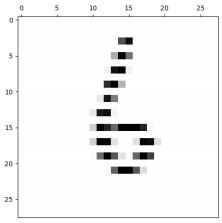
Table 7.5 shows that the neural network achieves the best classification accuracies, as expected, followed by the SPN algorithm. The neural network was also trained in around one hour, where it took over a 100 hours to train the SPN. The SPN algorithm has the best classification accuracies compared to the other tractable probabilistic models tested. Another interesting property of SPNs is that both the Naive Bayes model and GDA are essentially a special type of SPN. We now ask ourselves why we would not just always use a standard deep neural network over an SPN, as it has a better classification accuracy. To answer this question we need to remind ourselves that SPNs are probabilistic models. They can, therefore, perform queries that other neural networks cannot, e.g. easily work with missing inputs. We, therefore, evaluate this capability in the next section.

### 7.2.5 Capabilities of SPNs: Partial observations

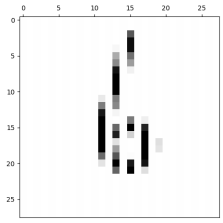
As observed in Section 7.2.4, we can see that vanilla feedforward neural networks seem to achieve the highest discriminative classification accuracies on the MNIST and Fashion MNIST datasets, given that we have provided all the inputs to the network. In real-world

applications, a machine learning system is often presented with incomplete data. Due to an SPN's probabilistic nature, it can easily work with missing data, without the need to be explicitly trained to do so. It is, therefore, of interest to see what the classification accuracies of SPNs are compared to neural networks when working with missing data.

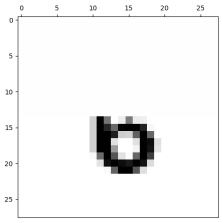
For our investigation, we use the MNIST dataset with partially marginalised out random variables, as can be seen in Figure 7.7.



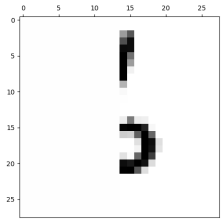
*(a) Horizontal obstruction*



*(b) Vertical obstruction*



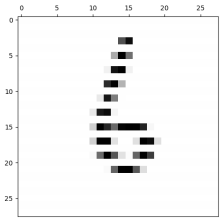
*(c) Top obstruction*



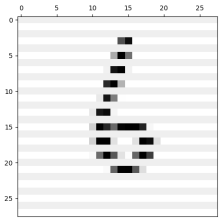
*(d) Left obstruction*

*Figure 7.7: Illustrations of different filters applied on MNIST digit images. These filters are used to indicate that information are hidden for the network. The four types of filters applied on the images are, removing every second row, shown in image (a), removing every second column, shown in image (b), removing the top half of the image, shown in image (c), and removing the left half of the image, as shown in image (d). Note the filter lines and background of each digit are both indicated in white. This is to simplify the illustration on how the partially observed images look. The white background is in fact represented with a zero value and the filtered pixels are unobserved.*

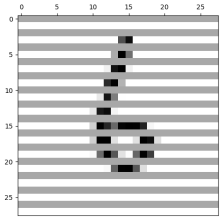
As there is no simple way of presenting a marginalised value to a neural network, we have to assign a specific value that represents marginalisation. We investigate four different colour schemes to indicate marginalisation to a neural network, as shown in Figure 7.8.



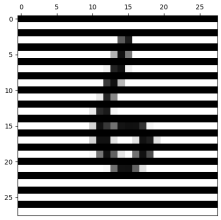
*(a) White filter*



*(b) Light grey filter*



*(c) Dark grey filter*



*(d) Black filter*

*Figure 7.8: Illustrations of different colour intensities for representing horizontal marginalisation on an MNIST image.*

By evaluating all four hypotheses, it was found that the neural network achieves the highest accuracies using the white filter. SPNs can easily handle this marginalisation by marginalising out the univariate leaf distributions for every unobserved random variable. Therefore, no special image processing needs to be performed beforehand. We first train both the feed-forward SPN and feed-forward neural network using the standard MNIST training set of 60 000 images, with 25% of it used for the validation set. Both models are then evaluated using the test set of 10 000 images. The neural network achieves an accuracy of 98.8%. The SPN achieves a final testing accuracy of 98.1%. We now investigate both networks' accuracies on the partially observed data. The results are presented in Table 7.6.

*Table 7.6: Test set accuracies of two network architectures, namely sum product network and deep neural network, on the MNIST test dataset. For both machine learning algorithms, a feed-forward architecture was used. Note that the neural network achieves a higher classification accuracy on the fully observed MNIST images. The SPN, however, does better when some of the image pixels are not given to the model. Here 'Full' indicates no filters are used. The categories 'Horizontal', 'Vertical', 'Top', and 'Left' represent the type of filter used, as specified in Figure 7.7.*

| ML algorithm | Full | Horizontal | Vertical | Top | Left |
|---|---|---|---|---|---|
| Sum Product Network | 98.1% | 94.2% | 94.3% | 70.2% | 62.2% |
| Neural Network | 98.8% | 92.3% | 91.1% | 55.1% | 42.2% |

From Table 7.6 one can see that the neural network achieves the highest classification accuracies on the fully observed images, but achieves a lower accuracy on partially observed data, than the SPN does. The neural network can possibly achieve a higher accuracy, by explicitly training on partially observed data, but this is, however, a complicated process as one does not usually know which variables will be unobserved. SPNs are, therefore, simple, but useful algorithms when dealing with partially observed data, while ensuring relatively high classification accuracies.

## 7.3 Conclusions

In this chapter, there was experimented with different training and inference capabilities of SPNs. We started by testing the compression capabilities of the Compress-SPN algorithm and found that it achieves a high compression percentage in a short amount of time. This algorithm is, therefore, an effective algorithm to use in between training of networks. A more powerful compression algorithm, like the tSPN algorithm, can then be used after training is completed. We then compared the classification accuracy of the SET-SPN algorithm to other SPN training algorithms and found that it achieved the best discriminative accuracy on the MNIST test dataset. There were better discriminative accuracies report for gradient descent methods in training SPNs, however, we found that these methods require large networks to work and take a long time to train. We, therefore, propose the SET-SPN algorithm as an easy to implement algorithm, that works

with most datasets and produces relatively fast results. The networks that are produced by the SET-SPN algorithm, also have fast inference times due to their small sizes. After the SET-SPN algorithm has been created the weights of the network can then further be updated using the Adam optimiser, proposed for the RAT-SPN algorithm.

We also found that SPNs have the highest classification accuracies compared to the other probabilistic models we tested. Deep neural networks, however, still have higher classification accuracies on fully observed data that SPNs do. We also investigated SPNs modelling capabilities on partially observed input data. We found that SPNs achieved good classification accuracies, even compared to vanilla fully connected neural networks, when partial inputs are provided. This is due to the structure of an SPN that can inherently cope with missing input data.

# Chapter 8

# Conclusion

## 8.1 Training algorithms

In this work, we investigate a range of training algorithms for SPNs, as well as derive a new structure learning algorithm, which we call SET-SPN. We provide a comparative study on the most popular learning algorithms to evaluate which perform the best. We also show that the SET-SPN algorithm achieved competitive results while creating compact networks with fast inference times. In the discriminative training setting the SET-SPN algorithm achieved the highest classification accuracy on the MNIST test dataset, compared to the other SPN training algorithms. For the generative training setting the SET-SPN algorithm achieved the second-highest testing accuracy for SPN training algorithms. Although there have been higher reported results for SPNs on the MNIST dataset, the SET-SPN algorithm creates more compact networks while still having relatively high accuracies. This greatly increases the inference speeds of networks, produced by the SET-SPN algorithm, compared to that produced by the other tested learning algorithms.

In this work, we also test the classification accuracies of SPNs compared to a neural network and other tractable probabilistic models. We found SPNs to achieve the highest classification accuracies, compared to the tractable probabilistic models we investigated. As expected, the neural network did have a higher classification accuracy on fully observed data than our SPN had. However, when working with partially observed data it was found that SPNs outperformed a standard feed-forward neural network.

Lastly, we derived a new fast compression algorithm called Compress-SPN. This com-

pression algorithm is designed to need only one pass through the network to operate. We designed the algorithm in this way so that a network can be compressed in a short amount of time. The Compress-SPN algorithm is useful in cases where fast network compressions are needed, as is needed in between anytime structure learning routines.

## 8.2 Inference capabilities

We also investigated different inference capabilities that SPNs have in comparison with neural networks and other probabilistic models. We found that SPNs have inference times linear in the size of the network. This means that all joint, marginal and conditional probability queries can be answered in a reasonable time, no matter what the network configuration is, barring the network is not too large. This is an attractive property that many of the more expressive probabilistic models do not have. We also derived a simple explainability algorithm called Explain-SPN specifically for SPNs, which helps to partially explain the network's predictions. In Section 4.7.1, we showed that SPNs can generate new data points. This can allow a person to better understand what features the SPN is looking for, for example, what pixels the network deems important for each digit class in the case of MNIST. This interpretability has the potential to allow experts to build more trust in these complex systems and better integrate these machine learning models into their professions.

## 8.3 Final conclusions

In conclusion, we find that SPNs are an attractive new type of hybrid model between deep neural networks and probabilistic models. SPNs have fast and exact inference, as neural networks do, but they still have the full range of probabilistic inference capabilities of a probabilistic model. This is due to an SPN, as with neural networks, approximating the data it is trained on and not approximating its inference, as some other probabilistic models do. The probabilistic properties of SPNs make them robust in the event of missing inputs and allow them to answer a wide range of probabilistic queries, which other neural network architectures are not capable of. This flexibility in machine learning algorithms is becoming increasingly more important as they become more integrated into our society.

SPNs, therefore, provide a promising new option in this pursuit for better algorithms to drive our world forward.

## 8.4 Future improvements

In future work, we would like to investigate applying the SPN algorithm to a real-world problem, where interpretability and flexibility of inference are important factors. We would like to focus on creating algorithms that can generate more complex explanations and therefore be more interpretable by experts. This will also direct the algorithmic development of SPNs to provide solutions to practical problems.

To apply SPNs on practical problems we need to scale them to work with larger datasets and decrease their training times. It is therefore of interest to try and train SPNs on large datasets, like the ImageNet dataset. In this work, we also derived an algorithm to deduce what the equivalent network is that each node observes. Therefore, the output of the network can be predicted, for changes in the node's weights, using only these equivalent networks. It is therefore of interest to see if we can use these equivalent networks to decrease weight tuning times and even avoid the problem of vanishing gradients. It might be possible that there are faster training algorithms for SPNs than just using the vanilla neural network methods.

To avoid overfitting it is also of interest to see if we can use the parameter to data point ratio and other methods, like better priors, to better predict when the network starts to overfit. Lastly, because we are working with a probabilistic model, we would also like to improve on the Explain-SPN algorithm to provide more reliable and better explanations to the predictions the network makes. It would also be of interest to apply the Explain-SPN algorithm on other datasets and investigate what explanations it generates.

# Appendix A

# Gradient derivations

In this section we derive the gradient values $\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})}$ and $\frac{\partial \ln(p_{ij})}{\partial w_{ijk}}$ for the sum nodes in an SPN. We start by deriving the gradient value $\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})}$, which is the gradient value of a sum node, at indices $(i+1)j_p$ ,with respect to a node in the previous layer, at indices $ij_c$. As defined in Section 5.2.1.2, the vector $\mathbf{c}(i+1, j_p)$ contains all the $j$ indices of the children of node $(i+1)j_p$. The first entry of this vector is represented by $c(i+1, j_p)_1$. As stated in equation (5.22), the output of a sum node can be written as

$$\begin{aligned} \ln(p_{(i+1)j_p}) &= \ln[e^{\ln(w_{(i+1)j_p1})+\ln(p_{ic(i+1,j_p)_1})-M} + \cdots + e^{\ln(w_{(i+1)j_pK})+\ln(p_{ic(i+1,j_p)_K})-M}] + M \\ &= \ln[e^{\ln(w_{(i+1)j_p1})+\ln(p_{ic(i+1,j_p)_1})} + \cdots + e^{\ln(w_{(i+1)j_pK})+\ln(p_{ic(i+1,j_p)_K})}]. \end{aligned} \tag{A.1}$$

The $M$ value is only included in equation (A.1) to make the computation numerically stable using 64-bit floats. We can, therefore, remove this symbol, as is done in the last line of equation (A.1), and then just make sure the resulting gradient values can be calculated in a numerically stable manner.

## A.1 Sum child gradients

We now derive the gradient of a sum node, with indices $(i+1)j_p$, with respect to a node in the previous layer as

$$\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})} = \begin{cases} \frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})} & j_c \text{ in } \mathbf{c}(i+1, j_p) \\ 0 & \text{Otherwise} \end{cases}. \tag{A.2}$$

Equation (A.2) is true, because two nodes, in two consecutive layers, which are not directly connected to each other have a gradient value of zero. We again define $c_p(i+1, j_p, j_c)$ as the index value in vector $\mathbf{c}(i+1, j_p)$ where the entry value $j_c$ is located. This function, therefore, gives the index of the weight connecting the parent node with its child node. We now focus on deriving $\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})}$ for children of a sum node, e.g. $j_c$ in $\mathbf{c}(i+1, j_p)$. This can be done using

$$\begin{aligned}
\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})} &= \frac{1}{p_{(i+1)j_p}} \frac{\partial p_{(i+1)j_p}}{\partial \ln(p_{ij_c})} \\
&= \frac{1}{e^{\ln(p_{(i+1)j_p})}} \frac{\partial e^{\ln(w_{(i+1)j_p c_p(i+1,j_p,j_c)}) + \ln(p_{ij_c})}}{\partial \ln(p_{ij_c})} \\
&= e^{-\ln(p_{(i+1)j_p})} e^{\ln(w_{(i+1)j_p c_p(i+1,j_p,j_c)}) + \ln(p_{ij_c})} \\
&= e^{\ln(w_{(i+1)j_p c_p(i+1,j_p,j_c)}) + \ln(p_{ij_c}) - \ln(p_{(i+1)j_p})}.
\end{aligned} \tag{A.3}$$

We express equation (A.3) in this form as it allows for small probability values to be manipulated in a 64-bit float, in the more stable logarithmic domain, before the exponent is taken of the result. Adding this gradient value back into equation (A.2) gives us

$$\frac{\partial \ln(p_{(i+1)j_p})}{\partial \ln(p_{ij_c})} = \begin{cases} e^{\ln(w_{(i+1)j_p c_p(i+1,j_p,j_c)}) + \ln(p_{ij_c}) - \ln(p_{(i+1)j_p})} & j_c \text{ in } \mathbf{c}(i+1, j_p) \\ 0 & \text{Otherwise} \end{cases}. \tag{A.4}$$

## A.2 Sum weight gradients

We now derive the gradient values of a sum node at indices $ij$ with respect to one of its weights ($\frac{\partial \ln(p_{ij})}{\partial w_{ijk}}$) as follows

$$\begin{aligned}
\frac{\partial \ln(p_{ij})}{\partial w_{ijk}} &= \frac{1}{p_{ij}} \frac{\partial p_{ij}}{\partial w_{ijk}} \\
&= \frac{1}{e^{\ln(p_{ij})}} \frac{\partial e^{\ln(w_{ijk}) + \ln(p_{(i-1)c(i,j)_k})}}{\partial w_{ijk}} \\
&= e^{-\ln(p_{ij})} \frac{1}{w_{ijk}} e^{\ln(w_{ijk}) + \ln(p_{(i-1)c(i,j)_k})} \\
&= \frac{1}{w_{ijk}} e^{\ln(w_{ijk}) + \ln(p_{(i-1)c(i,j)_k}) - \ln(p_{ij})} \\
&= \frac{1}{w_{ijk}} \frac{\partial \ln(p_{ij})}{\partial \ln(p_{(i-1)c(i,j)_k})}.
\end{aligned} \tag{A.5}$$

# Appendix B

# Computing specifications

In Table B.1 the specifications of important hardware components are listed.

*Table B.1: This table contains specifications of computer hardware used to run the experiments in this thesis.*

| Hardware type | Name | Specifications |
|---|---|---|
| CPU | Intel i7-3970X | The CPU has a clock speed of 3.50GHz with 12 cores and 24 threads. |
| GPU | Nvidia Titan Xp | The GPU has 12 GB internal memory with a base clock speed of 1.4 GHz. |
| RAM | - | The RAM component has 64 GB system memory with a transfer speed of 1.33 GT/s. |

# Appendix C

# Investigation on weight convergence

The Expectation Maximisation algorithm, derived for SPNs, is a generative algorithm with a relatively fast convergence rate, due to the algorithm's ability to dynamically choose its weight update step size. It is, therefore, of interest to do a simple investigation on the convergence of the EM algorithm, compared to the gradient descent algorithm, for weight tuning in the SET-SPN structure learner.

In the SET-SPN algorithm, 3 possible types of expansions can be made, assuming continuous variables are defined over Gaussian leaf distributions. The network can either expand a product node with Gaussian children, sum children or a mixture of the two. For illustrative purposes, we demonstrate the results for expanding two Gaussian children nodes. Most expansions are performed on product nodes with two Gaussian children nodes, due to a large number of such product nodes. We investigate both the EM and gradient descent algorithm on a generative toy problem. The toy dataset used in our investigation can be seen in Figure C.1.

The toy distribution in Figure C.1 can be modelled with two multivariate Gaussian distributions, with diagonalised covariance matrices, and thus each Gaussian distribution is represented by a product of two univariate Gaussians. We have, therefore, not made it hard to model this distribution, as we are simply interested in illustrating convergence speeds. For this task we created an SPN that we know can represent this data and the challenge imposed is to learn the parameters of the distribution. To represent this dataset we create an SPN that represents a Gaussian mixture model. This SPN represents how a single product node, with Gaussian children, would look like after the MIXEDCLONES algorithm was applied to it.

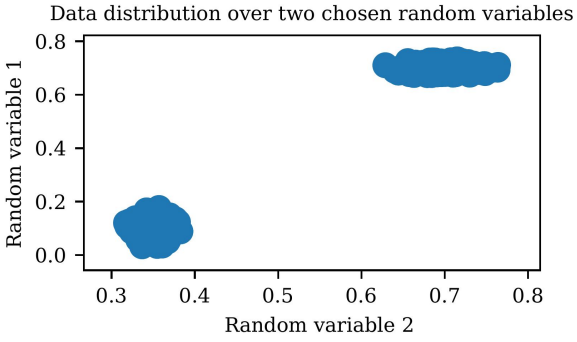



*Figure C.1: Generated toy dataset over two random variables. The data is generated from two Gaussians with different means and covariances.*

This Gaussian mixture model can fully represent the real data distribution. The Gaussian SPN, tasked with representing this distribution, is illustrated in Figure C.2.

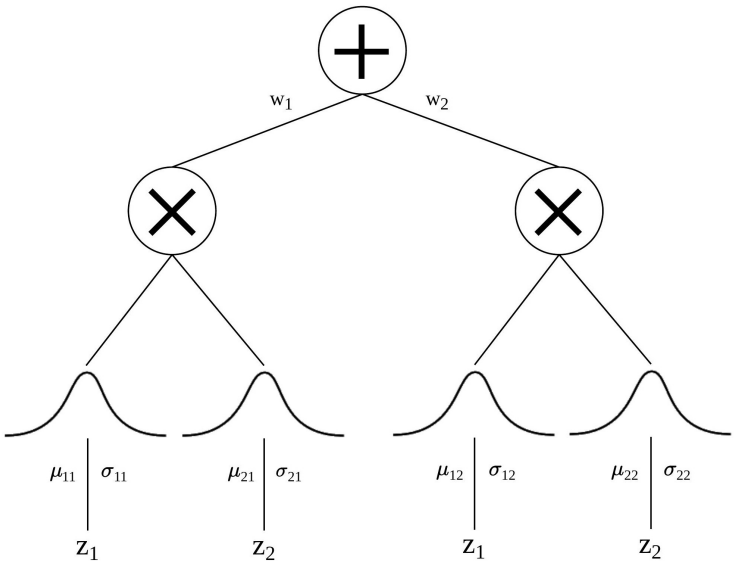
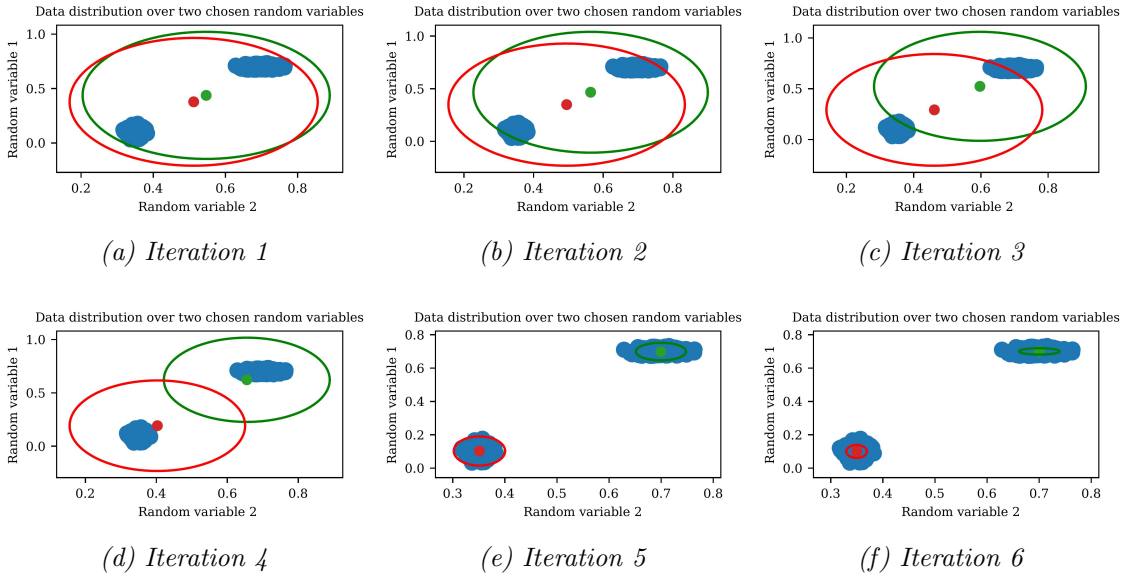



*Figure C.2: An SPN equivalent of a Gaussian Mixture Model (GMM) with two mixture components. Note that the mixture model is the product of two univariate Gaussians, instead of one multivariate Gaussian, because the data is generated from such a distribution.*

We investigate the EM algorithm first on this toy dataset. Figure C.3, illustrates the results of applying the EM algorithm on this toy dataset. The EM algorithm quickly

converges to approximately the correct answer in about six iterations. This quick convergence is in part due to the algorithm not needing a learning rate. There is, therefore, no added complexity of assigning how fast the algorithm should update its values.

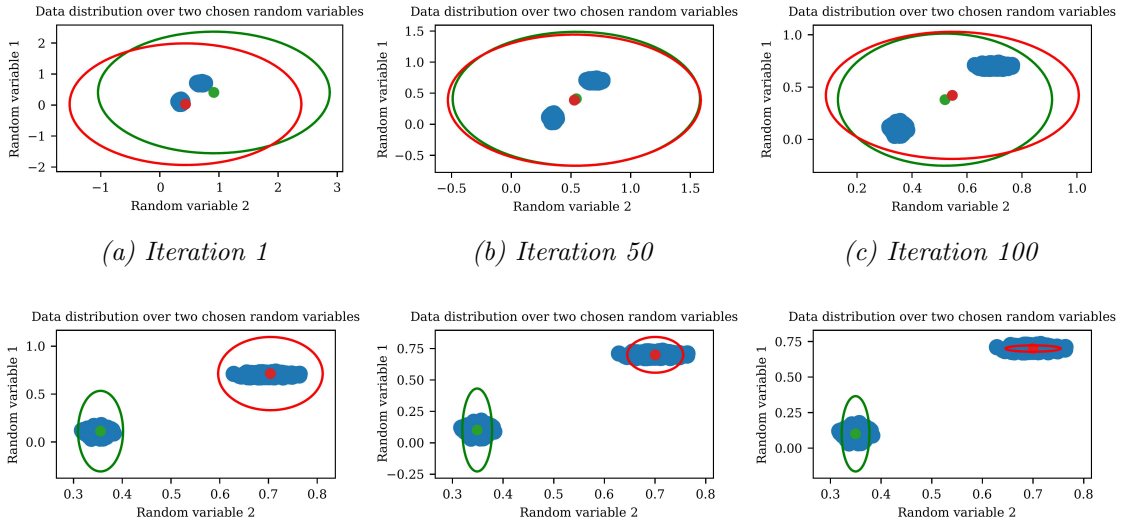



*(a) Iteration 1* *(b) Iteration 2* *(c) Iteration 3*

*(d) Iteration 4* *(e) Iteration 5* *(f) Iteration 6*

*Figure C.3: Illustrations of outputs for 6 iterations of the EM algorithm on the toy dataset. Note that as the EM algorithm updates the weights in the network, both the mean and the variance change.*

We also investigate a gradient descent optimiser, called the Adam optimiser, on this dataset. The Adam optimiser, however, requires quite a few more iterations before it starts to converge on a given answer. Figure C.4 show six iterations spread over the first 250 iteration steps. The gradient descent optimiser, therefore, needs a larger number of iterations before it converges to a suitable solution. Both the gradient descent optimiser and the EM optimiser take roughly the same amount of time to complete one iteration. One also needs to assign a learning rate to the algorithm, which may be hard to estimate.




*(a) Iteration 1* *(b) Iteration 50* *(c) Iteration 100*

*(d) Iteration 150* *(e) Iteration 200* *(f) Iteration 250*

*Figure C.4: Different iteration outputs, spread over 250 iterations, using the Adam gradient descent optimiser on the toy dataset. The learning rate was chosen to be 0.01 as it was found to allow for fast convergence for this given problem. Note that between each consecutive image (except between the first and second iteration) 50 iterations have elapsed.*

Table C.1 shows the results obtained for training three different algorithms on five different toy datasets. Some of the toy datasets also have overlapping data between mixture components. We also ran the basic gradient descent update rule, defined in equation (5.12), on these datasets.

*Table C.1: Average training times and accuracies of three different learning algorithms on five different toy datasets (some overlapping and some not), with one of these datasets illustrated in Figure C.1. The algorithms' average performance over 50 runs (10 per toy dataset) is tabulated. Each algorithm is allowed to run for 10 seconds. The Gaussian mixture model, which the data was generated from, has a log-likelihood score of 18 655 on the data itself. Note that, because we are working with density functions the log-likelihood of the data given a model can be greater than 0.*

| Algorithm | Train time (seconds) | Log likelihood (Higher is better) |
|---|---|---|
| Expectation Maximisation | 10.0 | 18 721 |
| Basic Gradient Descent | 10.0 | 10 834 |
| Adam Optimiser | 10.0 | 14 412 |

It seems that the EM algorithm generally outperforms gradient descent in this simple generative domain. The EM algorithm even gets a better log-likelihood score than the Gaussian mixture model that the data was generated from. This seems to indicate that the network slightly overfitted to the noisy data provided, as the network predicts the data is slightly more likely than it actually is. However, it is important to note that the Adam optimiser usually does better on more complex data distributions. This is evident in Section 7.2.2.1, were we found the Adam optimiser outperformed both standard gradient descent and the EM algorithm.

# References

Adel, T., Balduzzi, D. & Ghodsi, A. (2015). Learning the structure of sum-product networks via an svd-based algorithm. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence*, UAI'15, 32–41, AUAI Press, Arlington, Virginia, United States. 7, 20, 21

Cory J. Butz, A.E.d., Jhonatan S. Oliveira & Teixeira, A.L. (2019). Deep convolutional sum-product networks. vol. Vol 33 No 01: AAAI-19, IAAI-19, EAAI-20. 18

Dennis, A. & Ventura, D. (2015). Greedy structure search for sum-product networks. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, 932–938, AAAI Press. 22, 83

Dennis, A. & Ventura, D. (2017a). Autoencoder-enhanced sum-product networks. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 1041–1044. 53

Dennis, A. & Ventura, D. (2017b). Online structure-search for sum-product networks. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, 155–160. 6

Desana, M. & Schnörr, C. (2016). Expectation maximization for sum-product networks as exponential family mixture models. Heidelberg University, Institute of Applied Mathematics, Image and Pattern Analysis Group Im Neuenheimer Feld 205 69120 Heidelberg. 6, 78

Frankle, J. & Carbin, M. (2018). The lottery ticket hypothesis: Training pruned neural networks. *CoRR*, **abs/1803.03635**. 108

Gens, R. & Domingos, P. (2012). Discriminative learning of sum-product networks. In *Advances in Neural Information Processing Systems 25*, 3239–3247, Curran Associates, Inc. 6, 17, 46

Gens, R. & Domingos, P. (2013). Learning the structure of sum-product networks. In *Proceedings of the 30th International Conference on Machine Learning*, vol. 28 of *Proceedings of Machine Learning Research*, 873–880, PMLR, Atlanta, Georgia, USA. 19, 103

Hinton, G.E., Osindero, S. & Teh, Y.W. (2006). A fast learning algorithm for deep belief nets. *Neural Comput.*, **18**, 1527–1554. 2

Hsu, W., Kalra, A. & Poupart, P. (2017). Online structure learning for sum-product networks with gaussian leaves. *ArXiv*. 6

Jaini, P., Ghose, A. & Poupart, P. (2018). Prometheus : Directly learning acyclic directed graph structures for sum-product networks. In *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*, vol. 72 of *Proceedings of Machine Learning Research*, 181–192, PMLR, Prague, Czech Republic. 5, 24

Kalra, A., Rashwan, A., Hsu, W.S., Poupart, P., Doshi, P. & Trimponias, G. (2018). Online structure learning for feed-forward and recurrent sum-product networks. In *Advances in Neural Information Processing Systems 31*, 6944–6954, Curran Associates, Inc. 6, 38

Kingma, D.P. & Ba, J. (2015). Adam: A method for stochastic optimization. *CoRR*, **abs/1412.6980**. 76, 98

Ko, C.Y., Chen, C., Zhang, Y., Batselier, K. & Wong, N. (2018). Deep compression of sum-product networks on tensor networks. *ArXiv*, **abs/1811.03963**. 25, 97

Koller, D. & Friedman, N. (2011). *Probabilistic Graphical Models : Principles and Techniques*. MIT Press Ltd. 13

Krizhevsky, A., Sutskever, I. & Hinton, G.E. (2012). Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, 1097–1105, Curran Associates Inc., USA. 2

Kumar, A. & V, K. (2017). A study on system reliability in weibull distribution. *International Journal of Innovative Research in Electrical, Electronics, Instrumentation and Control Engineering*, **3297**, 2321–2004. 109

Levray, A. & Belle, V. (2019). Learning tractable probabilistic models in open worlds. University of Edinburgh and Alan Turing Institute, London. 3

Lowd, D. & Domingos, P.M. (2012). Learning arithmetic circuits. *CoRR*, **abs/1206.3271**. 49

Montavon, G., Samek, W. & Muller, K. (2018). Methods for interpreting and understanding deep neural networks. *Digital Signal Processing: A Review Journal*, **73**, 1–15. 2

Peharz, R., Vergari, A., Stelzner, K., Molina, A., Trapp, M., Kersting, K. & Ghahramani, Z. (2018). Probabilistic deep learning using random sum-product networks. *CoRR*, **abs/1806.01910**. 5, 6, 17, 105

Poon, H. & Domingos, P. (2011). Sum-product networks: A new deep architecture. In *Proceedings of the IEEE International Conference on Computer Vision*, University of Washington Seattle, WA 98195, USA. 15, 16, 17, 18, 26, 68

Rahman, T. & Gogate, V. (2016). Merging strategies for sum-product networks: From trees to graphs. In *Proceedings of the Thirty-Second Conference on Uncertainty in Artificial Intelligence*, UAI'16, 617–626, AUAI Press, Arlington, Virginia, United States. 25, 89

Rashwan, A., Poupart, P. & Zhitang, C. (2018). Discriminative training of sum-product networks by extended baum-welch. In *Proceedings of the Ninth International Conference on Probabilistic Graphical Models*, vol. 72 of *Proceedings of Machine Learning Research*, 356–367, PMLR, Prague, Czech Republic. 6

Van de Wolfshaar, J. & Pronobis, A. (2019). Deep convolutional sum-product networks for probabilistic image representations. *ArXiv*, **abs/1902.06155**. 5, 10, 18, 47, 67, 105

Zhou, Y. (2011). Structure learning of probabilistic graphical models: A comprehensive survey. Michigan State University, 220 Trowbridge Rd, East Lansing, MI 48824, USA. 14